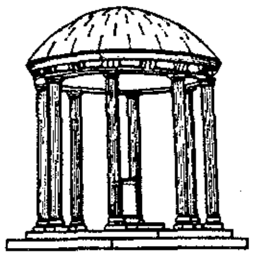
An Interface Description Language Assertion Checker

TR86-014

August 1986

Jerry Kickenson

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



AN INTERFACE DESCRIPTION LANGUAGE ASSERTION CHECKER

TR86-014

by
Jerry Scott Kickenson

A Thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill, 1986

Approved by:

[illegible]
Adviser

[illegible]
Reader

David A. Plaisted
Reader

JERRY SCOTT KICKENSON. An Interface Description Language Assertion Checker. (Under the direction of RICHARD T. SNODGRASS.)

Abstract

The Interface Description Language (IDL) is a notation for describing abstract properties of data structures that are passed among cooperating programs. IDL allows a user to specify a data structure in terms of a graph of attributed nodes. A tool, the IDL translator, generates readers and writers that map between specific internal representations of the structure in a target programming language and the abstract external representation.

However, IDL has its shortcomings. The structuring component of IDL does not allow the user to specify certain properties of the abstract structure he or she might wish to specify, such as that the structure is a binary tree. In addition, it does not provide a means to specify constraints on the values of particular attributes of the nodes in the structure. The IDL assertion language allows the user to express such specifications and constraints. The IDL assertion checker is a tool that automatically checks that instances of an IDL structure satisfy the specifications and constraints expressed in the assertion language.

Our thesis is that a useful tool to verify assertions expressed in the assertion language can be implemented. This doucument and the implementation it describes are a demonstration of this thesis.

Contents

Part I

Introduction

CHAPTER 1

Introduction

The Interface Description Language (IDL) is a notation for describing the data structures passed between cooperating processes in a large system. IDL can define any structure that can be expressed as a graph. The language is at a higher level than conventional programming languages, and so is easier to understand. A tool, the IDL translator, maps the IDL descriptions into the equivalent code in a target programming language. The readers and writers necessary to map the IDL descriptions to and from the internal representation of the structures are automatically generated. In addition, the translator defines several macros to manipulate the data structures. These macros are framed in terms of the higher level IDL description, so the user need not think in terms of the specific programming language's view of the data. He may always view the data from the abstract IDL viewpoint.

IDL cannot describe all characteristics of a data structure. Features of the graph, such as that the graph is a tree, or that no node has more than a certain number of children, cannot be expressed in IDL. Specifying the values of attributes in the IDL graph cannot be done within IDL.

To allow the user of IDL more control over the structures he creates, the assertion language was proposed [4]. The language permits the user to express specifications about the IDL structure or the values of attributes within the structure. A tool, the assertion checker, can then automatically check the validity of these specifications on particular structure instances.

The assertion language is meant to serve as a specification language for IDL and as a debugging tool. After formally specifying the characteristics of the input and output structures of a process, the user would then run the process on a particular input instance, producing an output instance. The assertion checker would then be run on the instances, reporting on any assertions that were not satisfied and offering as much information as possible to aid the user in discovering why the assertion was not satisfied.

Formal verification of programs would be ideal. However, it is our position that formal verification of large programs is well beyond the state of the art. Our thesis is that a useful assertion checker tool is both feasible and practical. This document supports the thesis by describing a working implementation of an assertion checker. Efficiency was not a primary goal of the current implementation. We were most interested in demonstrating the feasiblility of the tool. However, we describe how the current implementation can be further improved and made more efficient.

1.1. Previous Work

IDL is a direct descendant of the Linear Graph package of the Production Quality Compiler-Compiler(PQCC) project at Carnegie-Mellon University. Lamb describes several other projects involved in intermediate representations or automatic generation of data structures. [4]

IDL itself was specified and partially implemented at Carnegie-Mellon University in 1981. It was used as the Diana intermediate representation of Ada [2] and has been employed in several industrial projects since 1981. An assertion checker was not part of any of these implementations.

More recently, an implementation of an IDL translator has been accomplished at the University of North Carolina at Chapel Hill.

As far as we know, an assertion checker of this type had not been written before. However, Lamb outlined possible data structures and algorithms a checker might use [4].

1.2. Outline of the Thesis

Part II is a tutorial introduction to the assertion language and its use. Part III describes the current implementation of the assertion checker, reports the measured performance of the checker, and recommends ways to improve the current implementation. Part IV concludes the document and suggests future research and work.

1.3. Conventions Used in This Document

Words or phrases that denote important concepts will be printed in *italics* on first appearance. These words and phrases are collected in an index at the end of the text and the page number of the definition appears in **bold** in the index.

All syntactic definitions are given in extended Backus-Naur Form (BNF). Angle brackets ("<>") surround the name of a non-terminal. Braces ("{}") are used to group elements of a production; a trailing asterisk ("*") following braces indicates zero or more occurrences; a trailing plus ("+") indicates one or more occurrences; a trailing question mark ("?") indicates an optional item. Reserved words are shown in **bold**. Examples are shown in `fixed width`.

Part II

Tutorial

CHAPTER 2

Introduction

2.1. Overview

The IDL Assertion Language is a sublanguage of IDL (Interface Description Language). It permits the user of IDL to make assertions about the IDL structures he has written and the values of attributes within those structures. A program called *idlcheck* can then automatically check the validity of these assertions on particular structure instances.

This document is a tutorial introduction to using the assertion language and *idlcheck*. First, basic features of the assertion language will be described, along with examples of their use. An explanation of how to use the assertion checker will be provided. Second, a complete example of a small IDL specification with relevant assertions will be presented. Some more advanced features will be introduced. The process by which the assertions were composed will be explained. Third, brief descriptions of the most advanced features of the assertion language will be presented. Finally, appendices contain the complete specification for the example presented in chapter 3 and several examples of assertion checker output.

This tutorial has been written for the practicing computer scientist. Experience with programming in some high level language is assumed. In addition, the reader should be familiar with IDL, at least to the level covered in *A Tutorial Introduction to Using IDL*[6]. The structure specification examples used in this tutorial are drawn from that document.

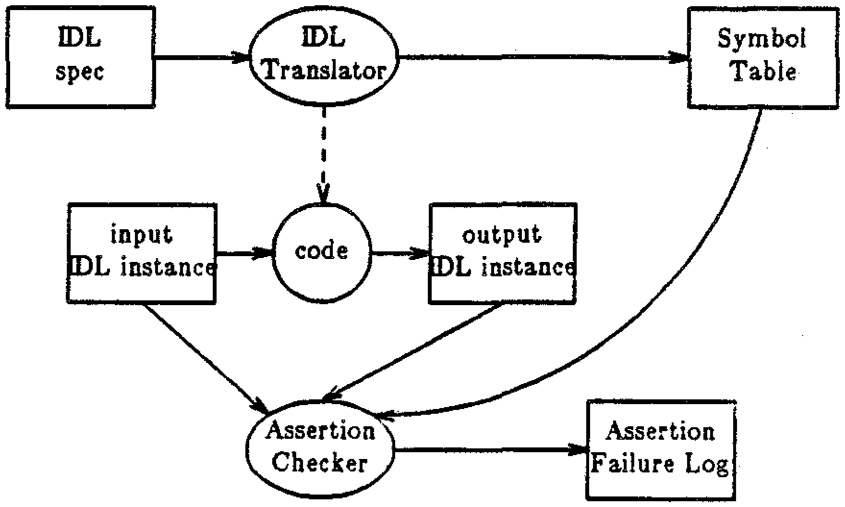
2.2. Purpose of Assertions

The assertion language is meant to provide both specification and verification facilities to IDL.

As a specification language, the assertions provide a means for the programmer to specify precisely what it is he wants to be true in his data structures. By using the assertion language, communication between members of a programming team or between future and present programmers is improved. Maintenance of large programs is eased, since the maintainer may look at the assertions to ascertain exactly what is supposed to be done, rather than attempting to figure this out by directly reading code or comments (which are often sparse, incomplete, and imprecise). In addition, writing assertions serves to hone the programmer's thinking about a particular problem, since to write significant assertions requires a thorough understanding of the problem. Finally, the assertions provide a guide to the writing of the code. Assertions are given in terms of the IDL structures. Programming in the IDL system involves manipulating these structures. Thus, well written assertions state precisely what changes of the IDL structures must be done by the code.

As a verification language, assertions provide a debugging aid. Once the assertions are written, instances of data structures can be checked automatically. No longer does the programmer have to scan manually through pages of output to ensure that there were no errors. Errors will be found and reported.

The following figure illustrates where the assertion checker tool fits in with the other IDL system tools. Rectangles denote data, ellipses denote executable code, and arrows denote input and output of IDL instances.



In the above figure, assertions are placed in the IDL specifications (IDL spec). The user writes some of the code (in particular, the implementation of the algorithm). Everything else is generated automatically. The translator produces much of the code and the Symbol Table. The Assertion Checker generates the Assertion Failure Log, which is a report of any user assertions found to be false.

CHAPTER 3

Basic Use of the Assertion Language

3.1. Example Specification

The following example will be used to illustrate use of the assertion language. The example is taken from [6].

```
--Specification for the customers structure.

Structure customers Root customer_list Is

    customer_list => list : Seq Of Customer;

    Customer ::= commercial_customer | Government_customer;
    Customer => name : String,
             address : String,
               active : Boolean,
               customer_number : Integer,
               balance : Rational;

    commercial_customer => industry_code : Integer;

    Government_customer ::= state_customer | federal_customer;
    state_customer      => state_code : Integer;
    federal_customer => agency_code : Integer;

End

--Specification for the transactions structure.

Structure transactions Root transaction_list Is

    transaction_list  =>  list : Seq Of Transaction;

    Transaction    ::=  credit | debit;
    Transaction     => customer_number : Integer,
                 date : Integer,
                 amount : Rational,
                 tax_status : Set Of Tax_code;

    credit => ;
    debit => ;

    Tax_code ::= local_sales_tax
```

```
                | state_sales_tax
                | federal_sales_tax;

    local_sales_tax => ;
    state_sales_tax => ;
    federal_sales_tax => ;

End


--Specification for the bills structure.

Structure bills Root bill_list Is

    bill_list => list : Seq Of bill;

    bill => billee : Customer,
          amount : Rational;

    Customer ::= commercial_customer | Government_customer;
    Customer =>   name : String,
            address : String,
            customer_number : Integer;

    commercial_customer => industry_code : Integer;

    Government_customer ::= state_customer | federal_customer;
    state_customer => state_code : Integer;
    federal_customer => agency_code : Integer;

End


--Specification for the billing process.

Process billing Inv billing Is

    Pre     customers_in : customers;
    Pre     transactions_in   : transactions;
    Post customers_out : customers;
    Post bills_out : bills;

End
```

3.2. Basic Assertions With Quantifiers

Assertions always begin with the keyword Assert. The simplest assertions to make concern the values of specified attributes within a structure. These assertions may appear anywhere within the structure to which they refer, although it is often clearer to group all assertions together. It is possible to assert a property about a single object, For example, to assert in the `customers` structure that the customer list is not empty, we could write:

```
Assert Size(Root.list) ~= 0;
```

Root refers to the root of the structure in which the assertion appears. Here, the root of the `customers` structure is of type `customer_list`. Thus, `Root.list` is a sequence of

Customer. The assertion states that the size of this sequence is not zero.

Usually one wishes to assert about all objects within a structure of a certain type. This is done using *quantifiers*. There are two variants of the quantifier. The **ForAll** variant asserts that the body is true for all objects within the structure in which the assertion appears, of the specified type. For example, we might wish to make some assertions about the customers structure. First, all customer numbers should be greater than or equal to one. A negative customer number would signify an input error. We would assert this as:

```
Assert ForAll C In Customer Do C.customer_number >= 1 Od;
```

The name C is an *iterator* that will take on successive values of the specified object type, which here is Customer. The body of the quantifier states that the customer_number attribute of C (C is of type Customer) is greater than or equal to one. The period (.) indicates the extraction of an attribute. (This parallels the => symbol in the IDL specification.) Since the **ForAll** variant is used, we are asserting that the customer_number attribute of all objects of type Customer have the described property.

The **Exists** variant asserts that the body is true for at least one object within the structure in which the assertion appears, of the specified type. For example, to assert that at least one customer is active, we could write:

```
Assert Exists C In Customer Do C.active = True Od;
```

The body of a quantifier, whether **ForAll** or **Exists**, is a boolean entity. Despite the use of the **Do...Od** form, nothing is *done*. We are asserting what must be true. Note also that the type of the iterator, specified immediately after the keyword **In**, must be a valid type in the IDL specification. A valid type is any node or class name (but not attribute names) or a basic type (**Integer, Rational, String, Seq Of** <type>, or **Set Of** <type>).

Boolean operators such as **And, Or,** and **Not** may be used. For example, we wish to assert that the state_code of every state_customer is between 1 and 50, inclusive. We could write:

```
Assert ForAll sc In state_customer Do
      sc.state_code >= 1 And
      sc.state_code <= 50
       Od;
```

Nesting of quantifiers is allowed. For example, we wish to assert that no two distinct customers have the same customer number. This could confuse billing and credits. We would assert as follows:

```
-- No duplicate customer numbers
Assert ForAll c1 In Customer Do
      ForAll c2 In Customer Do
            If c1 ~= c2 Then
                c1.customer_number ~= c2.customer_number
            Else True Fi
      Od Od;
```

The symbol ~= means "not equal to". The two iterator names must be distinct when using nested quantifiers. Here the names used are c1 and c2. **Od** appears twice at the end. Each occurrence of a quantifier must be ended by its own **Od**.

Similar assertions on attribute values might be made in the transactions structure and the bills structure. For instance, we could assert in the transactions structure:

```
Assert ForAll t In Transaction Do t.customer_number >= 1 Od;
```

In the `bills` structure, we might wish to be certain that the amount of all bills is positive. We could then assert in the `bills` structure:

```
Assert ForAll b In bill Do b.amount > 0 Od;
```

The assertion language supplies a few built-in functions that are useful. The function Size was used above. It returns the number of objects in a set, sequence or collection (described below) or the number of characters in a string. Other functions will be introduced below.

String constants are characters enclosed within quotes. For example, we wish to be certain that an important customer named `IBM` is in our customer list. We would then assert:

```
Assert Exists C In Customer Do C.name =" IBM" Od;
```

3.3. Assertions within Processes

The above assertions concern particular structures. They are thus placed inside the structure to which they refer. It is also possible to make assertions within process specifications. In this way, one can make assertions about the relation of output structures to input structures.

For example, in the process `billing`, there is an instance of the `customers` structure read in and an instance of the `customers` structure written out. The information within these instances should remain unaltered, except perhaps for the balance attribute of `Customer`. We could assert this as follows (within the process specification):

```
-- Customers output match customers input.
Assert ForAll c_in In customers_in:Customer Do
      ForAll c_out In customers_out:Customer Do
          If  c_in.customer_number  =  c_out.customer_number
          Then
              c_in.name = c_out.name And
              c_in.address = c_out.address And
              c_in.active = c_out.active
          Else True Fi
      Od Od;
```

The specified iterator types in the above quantifiers are prefixed with a port name followed by a colon. The prefix specifies the name of the port associated with the structure instance the assertion is referring to. In the above example, the first quantifier is referring to all objects of type `Customer` in the structure associated with the port named `customer_in`. The second quantifier is referring to all objects of type `Customer` in the structure associated with the port named `Customer_out`. These structures must both contain the type specified (here `Customer`). The use of these *port prefixes* is allowed only within a process specification, since structure specifications do not have ports. Clearly, a port with the name specified in the prefix should exist if the assertion is to make sense.

In the last example the `customer_number` and `balance` attributes were not mentioned. This is because the If condition guarantees that the `customer_number` attribute is equal, and the balance attribute may be legitimately altered owing to transactions being processed. In addition, the `industry_code`, `state_code`, and `agency_code` attributes were not mentioned. These should remain unaltered also, but they are not attributes of the entire class `Customer`. They are attributes of particular node members of that class. Thus statements about them cannot be made within the context of the entire class. For example, to assert:

```
-- Bad Assertion!
Assert ForAll c In Customer Do
      c.customer_number > 0 And
      c.industry_code > 0    Od;
```

would result in an error, because `c` is of type `Customer`, which does not necessarily have an `industry_code` attribute. If one wishes to make an assertion about such an attribute, one must write an assertion specifically targeted toward that class of node containing the attribute. For example, we could assert:

```
Assert ForAll cc_in In customers_in:commercial_customer Do
      ForAll cc_out In customers_out:commercial_customer Do
         If cc_in.customer_number = cc_out.customer_number
         Then
            cc_in.industry_code = cc_out.industry_code
         Else True Fi
      Od Od;
```

Since `commercial_customer` is a member of the `Customer` class, we can refer to the `customer_number` attribute of `cc_in` and `cc_out`, which are of type `Customer`. Since we have specified `cc_in` and `cc_out` as type `commercial_customer`, we can now also refer to an `industry_code` attribute. Similar assertions could be made about the `state_code` for objects of type `state_customer` and the `agency_code` for objects of type `federal_customer`.

As another example, we can assert that every transaction refers to a customer.

```
-- Each transaction refers to a customer
Assert ForAll t In transactions_in:Transaction Do
      Exists c In customers_in:Customer Do
          t.customer_number = c.customer_number
      Od Od;
```

3.4. Definitions

Some assertions we might wish to make can get more complicated than the examples above. To simplify the formation of these assertions, the assertion language allows one to create definitions, and then use these definitions in assertions.

3.4.1. Value Returning Definitions

The simplest definition returns a value. For example, we wish to assert that the balance of a customer on output from our billing process is equal to the balance the customer had on input plus any credit received through transactions. We are specifying the behavior of our program. The total credit a customer receives is the sum of all credit transactions for that customer. This suggests creating a definition that will take as arguments a customer and the list of transactions and return the total amount of credit for that customer. Such a definition could then be used in the assertion we wish to make about the customer's balance. The definition could take the following form:

```
Define Total_Credit(c:Customer, TList:Seq Of Transaction) =
      If Size(TList) = 0 Then 0
      OrIf Head(TList).customer_number = c.customer_number
      And
         Type(Head(TList)) Same transactions_in:credit Then
           Head(TList).amount + Total_Credit(c,Tail(TList))
      Else Total_Credit(c,Tail(TList))
      Fi;
```

The keyword Define must introduce any definition. The definition takes two arguments. The first is of type `Customer` and the second is a sequence of `Transaction`. If the size of the transaction sequence is 0, then there are no transactions. The definition then returns 0. Otherwise, if the transaction at the head of the sequence has the same customer number as the customer, meaning that this transaction deals with this customer, and the transaction is of type `credit` (the Type function returns the actual type of a class object) we add the amount of this transaction to the total credit of the remaining transactions (the **Tail** function returns the sequence that is the argument without its head). If the transaction does not concern this particular customer or the transaction is not a credit, then we return the total credit of the remaining transactions only.

The definition of `Total_Credit` is recursive. It is guaranteed to stop since the definition is applied to a smaller collection each time it is called (the **Tail** of a sequence is always smaller than the sequence itself, if the sequence is non-empty), and the definition does not call itself when an empty sequence (of size 0) is encountered.

The operator **Same** was used in the `Total_Credit` definition. When comparing two objects or values, the operator = is used. When comparing two collections of objects, the operator **Same** is used. The function **Type** returns the collection of all objects in the structure instance that are of the same type as the function argument. A class or node name returns the collection of all objects in the structure instance that are of the same type as the class or node. Thus in the definition of `Total_Credit`, **Type**`(`**Head**`(TList))` and `credit` both return collections of objects and must be compared with **Same** rather than =.

With this definition in hand, we can now make the assertion that the balance of a customer at output is the balance the customer had at input plus any credit it received.

```
-- All customers are properly credited.
Assert ForAll c_out In customers_out:Customer Do
       ForAll c_in In customers_in:Customer Do
          If  c_in.customer_number  =  c_out.customer_number
          Then
             c_out.balance = c_in.balance +
                Total_Credit(c_in,
             transactions_in:Root.list)
          Else True Fi
       Od Od;
```

The meaning of the above assertion should be clear except for two points. First, note the **Else True**. An **If** clause must have an Else clause attached. Here, if the customer numbers are not equal, then the program is fine. So we assert **True** in the **Else** clause. Second, the reserved word **Root** was used. **Root** refers to the root object of the specified structure. If no structure is specified, it refers to the root object of the structure in which the assertion or definition appears. Here, the port prefix `transactions_in` specifies we mean the root of the structure associated with the port `transactions_in`, which is the `transactions` structure. The root of this structure is of type `transaction_list`. This type has an attribute of type **Seq Of** `Transaction`. That is why the **Root**`.list` is specified as an argument. This dotted expression has type **Seq Of** `Transaction` since the specified root has an attribute called `list` that has this type.

In the `Total_Credit` definition, **OrIf** was used. The **OrIf** form is a shorthand that is recommended. Every use of **If** must be ended with a **Fi**. Use of **OrIf** does not require a **Fi**. Thus **OrIf** is preferable to Else If.

As another example of the use of definitions, we can use a definition to help us assert that the number of bills generated equals the number of debit transactions. We can create a definition that will return the number of debit transactions.

```
Define Num_debits(TList: Seq Of Transaction) =
       If Size(Tlist) = 0 Then 0
       OrIf Type(Head(TList)) Same transactions_in:debit Then
          1 + Num_debits(Tail(TList))
       Else Num_debits(Tail(TList)) Fi;
```

With this definition, we can now assert:

```
Assert Size(bills_out:Root.list) = Num_debits(transactions_in:Root.list);
```

Assertions may often be written in several ways. For instance, the above assertion could have been written without using the `Num_debits` definition:

```
Assert Size(bills_out:Root.list) =
            Size(transactions_in: debit);
```

3.4.2. Collection Returning Definitions

Definitions may also return collections of objects as opposed to values. Such definitions can prove to be useful. For instance, one may define a collection of objects not explicitly defined in the IDL structure, and then make some assertion about the objects in the collection. For example, we wish to assert that a bill is generated for each debit transaction. First we can define a definition that returns the collection of all objects that are debit transactions. Then we can assert that there exists a bill for each of these transactions.

```
Define Debits(TList:Seq Of Transaction) =
      If Size(TList) = 0 Then Empty
      OrIf Type(Head(TList)) Same transactions_in:debit Then
         Head(TList) Union Debits(Tail(TList))
      Else Debits(Tail(TList)) Fi;
```

The usual set operations (including **Union**, used above) are available to use with collections of objects that exist explicitly in the IDL structure specification or that are defined in the assertion language. If the transaction sequence is empty, the definition returns **Empty**. This denotes the empty collection, or the collection of no objects. Returning 0 would not make sense in this context, since the definition is returning a collection of objects.

We may now make our assertion:

```
-- A bill is generated for every debit transaction
Assert ForAll deb In Debits(transactions_in:Root.list) Do
      Exists b In Members(bills_out:Root.list) Do
         b.billee.customer_number = deb.customer_number And
         b.amount = deb.amount
      Od Od;
```

The above assertion makes use of the collection returning definition, and the two level deep dotted expression. `b.billee` is of type `Customer`. Thus, `b.billee` has an attribute called `customer_number`, and the reference to `b.billee.customer_number` is permitted. **Members** is a supplied function that takes a set or sequence (as here) and produces the collection containing all objects in that set or sequence. Use **Members** whenever you are interested in the objects contained in a set or sequence, rather than the set or sequence as a single object.

With the `Debits` definition above, there is yet another way to assert that the number of bills equals the number of debit transactions:

```
Assert Size(bills_out:Root.list)
      = Size(Debits(transactions_in:Root.list));
```

3.5. Use of the Assertion Language within Unix

There are three programs you need to be concerned with. They are *idlc*, *assertcodegen*, and *idlcheck*.

idlc is the IDL translator. Once you have written the IDL specifications for your process, use *idlc* to create the necessary source and object files, including the IDL *symbol table*. The symbol table is produced by the IDL translator with the -s option. It contains all the semantic information contained in the original IDL specification, but in a structured form. The symbol table is used by *assertcodegen* to generate code that *idlcheck* will interpret. *idlcheck* takes as input instances of the input and output of a particular run of the user's process as well as the code generated by *assertcodegen*, and produces an error log. This error log will report any assertions that were false, as well as information about the values of the expressions leading to the false assertion.

The IDL specifications for the billing process described above are in a file called `billing.idl`. The Unix command:

```
idlc billing.idl -s billing.symboltable
```

will create the IDL source (`billing.h`) and object (`billing.o`) files the user will link with his algorithm (`billing.algorithm`) , as well as the symbol table (`billing.symboltable`) needed by the assertion code generator. Next, the command:

```
assertcodegen billing.symboltable -o billing.check
```

will create the code that the assertion checker will interpret and place it into the file `billing.check`. Assuming the linked user's process code is in `billing`, the user will run `billing` on a particular input data instance (here, a customer list and a transaction list), creating output (here, an updated customer list and a billing list). Each separate structure must be placed in its own file. If the input customer list is in the file `in1`, the input transaction list is in the file `in2`, the output customer list is in the file `out1`, and the output billing list is in the file `out2`, then the final Unix step is:

```
idlcheck customers_in:in1 transactions_in:in2
        customers_out:out1 bills_out:out2 billing.check -e error_log
```

The names before the colon for each file is the name of the associated port as declared in the IDL specification for the `billing` process.

The assertion checker will check the assertions on the particular input and output indicated and write the error log out to the file `error_log`.

The input and output must be in the ASCII External Representation Language. This is automatically the case when working within the IDL system.

The above process may be abbreviated with Unix pipes. Thus, the three command steps may be condensed into:

```
idlc billing.idl -s -  | assertcodegen  | idlcheck
        customers_in:in1 transactions_in:in2
        customers_out:out1 bills_out:out2 billing.check -e error_log
```

If you do not make any assertions about a structure, then the file associated with that structure need not be present. At least one file must be present, or why are you checking assertions?

More information on the Unix IDL commands may be found online in the manual pages. The pages of interest are IDLC(1), ASSERTCODEGEN(1), and IDLCHECK(1) (see appendix D).

CHAPTER 4

Full Example

In this chapter, another IDL structure and process specification will be given. Various assertions will be written. The basic ideas will be reviewed, as well as some more advanced ideas.

4.1. Example Specification

The following is a specification that will be used to provide examples of assertions. The structure is taken from [6].

The structure represents the parse tree of one integer-valued function. The process will fold constants within the body of the function, and output the function with constants folded. The specification follows:

```
-- Specification for the constant folding process
Structure parse_tree Root func Is

    func => name : String,
          params : Seq Of formal,
          def     : Exp;

    formal => name : String;

    Exp      => const | formal | Operation;

    Operation ::= bin_op | un_op;
    Operation => op : Operator,
    left : Exp;

    bin_op => right : Exp;
    un_op => ;

    Operator ::= plus | minus | times | div;
    plus =>; minus =>; times =>; div =>;

    const => val : Integer;

End

Process constant_fold Inv parse_tree Is

    Pre     input :     parse_tree;
    Post output   : parse_tree;

End
```

4.2. Overloaded Definitions

First, we wish to assert that the structure output is actually folded. We make the following assertion:

```
Assert Folded(output:Root.def);
```

The prefix `output` before Root indicates that we are asserting about the root of the structure at the port called `output`, and not about the root of all structures.

In order for the definition to make sense, we must define what we mean by *Folded*. The following overloaded definition of *Folded* does the trick:

```
        Define Folded(e:const) = True;
```

If the expression is just a constant, then it is clearly folded.

```
        Define Folded(e:formal) = True;
```

If the expression consists of only a formal name, then we say it is folded.

```
        Define Folded(e:un_op) =
                If Type(e.left) Same output:const Then False
                Else Folded(e.left) Fi;
```

If the argument of a unary operator is a constant, then the expression is not folded, since the constant could be operated on to produce a single constant. So if the type of the argument is a constant, the expression is not folded. If the argument is not a constant, then the unary expression is folded if its argument is folded.

```
        Define Folded(e:bin_op) =
                If Type(e.left) Same output:const And
                        Type(e.right) Same output:const
                Then False
                Else Folded(e.left) And Folded(e.right)
                Fi;
```

In a binary expression, if the operands are both constants, then the expression is not folded, since the operands could have been operated upon to produce a single constant expression. Otherwise, the binary expression is folded if both its operand expressions are folded.

Folded must be defined for each kind of expression. Thus *Folded* is overloaded. By overloaded, we mean allowing a single function name have different meanings that are dependent on the number and type of its arguments. Each instance of the function takes care of one possible kind of expression. *Folded* is called in the assertion with the argument Root.def . The type of this argument is `Exp`. Since every node type in the `Exp` class (`const`, `formal`, `un_op`, and `bin_op`) appears as a formal argument of an instance of *Folded*, all is well. Overloaded definitions allow one to use the same name for a related group of definitions that serve the same purpose, but for different types of arguments.

The use of an overloaded definition is preferable to writing one definition instance that contains If clauses. One might have defined *Folded* as:

```
-- This is bad style!
Define Folded(e:exp) =
       If Type(e) Same const Then True
       OrIf Type(e) Same formal Then True
       OrIf Type(e) Same un_op Then ...
       Else ...
       Fi;
```

This method is more inefficient than the use of overloaded definitions. In addition, it can lead to semantic errors if the translator is unable to determine for certain the types of certain constructs, which is more often the case within If clauses. To avoid this worry, and to gain efficiency, use overloaded definitions rather than If clauses.

Next, we would like to assert that in the constant folding process, the value of the function has not been inadvertently changed. So we assert:

```
Assert Value(input:Root.def) = Value(output:Root.def);
```

It follows that we must define what we mean by the value of an expression. As above, we use an overloaded definition. Each instance will take care of one type of expression.

```
Define Value(e:const) = e.val;
```

The value of a constant is stored in its `val` attribute.

```
Define Value(e:formal) = 1;
```

The value of a `formal` node is set to 1. This is done because the value of a formal name is not known. It is just required that whatever value we assign to it, that value does not make the value of an expression the formal name is contained in equal to the value of another expression it would not normally be equal to. Any integer except zero would do.

```
Define Value(e:un_op) =
       If Type(e.op) Same input:plus
       Then Value(e.left)
       Else -1 * Value(e.left)
       Fi;
```

The value of a unary expression is the value of the unary operator's argument if the operator is unary plus. Otherwise the operator must be unary minus, and the value of the expression is -1 times the value of the argument.

```
Define Value(e:bin_op) =
       If Type(e.op) Same input:plus
            Then Value(e.left) + Value(e.right)
       OrIf Type(e.op) Same input:minus
            Then Value(e.left) - Value(e.right)
       OrIf Type(e.op) Same input:times
            Then Value(e.left) * Value(e.right)
       Else Value(e.left) / Value(e.right)
       Fi;
```

The value of a binary expression is the value resulting from applying the particular binary operator to the values of the operands.

The above definition of `Value` is recursive in two of its instances. It is guaranteed to stop since all expressions will eventually evaluate to a constant or formal expression. The instances of `Value` for these types are not recursive.

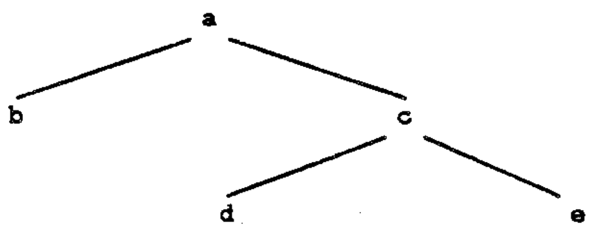
4.3. Cyclic Definitions

The function parse trees that the constant folding process takes as input and produces as output must be trees. However, the IDL specification for the parse tree does not restrict the structure to a tree. The `Operation` class has an attribute `left` that is of type `Exp`. It is not stated in the specification that the particular `Exp` pointed to by this attribute cannot be an instance of `Exp` closer to the parse tree root or on the other side of the root. In other words, cycles and cross branches are permitted, if not intended. In effect, there is nothing to prevent the parse "tree" from not actually being a tree. Such a strange instance of a parse tree could cause problems. Thus we wish to assert that the `parse_tree` structure is a tree.

Forming a precise assertion that a graph fulfills the requirements of a tree is not a trivial task. A few definitions will be useful. First, the immediate descendants of a node will be defined. This definition will take as an argument a node of type `Exp`. Since `Exp` is a class, and different kinds of `Exp` have different definitions of immediate descendants, an overloaded definition is called for. `Exp` nodes of type `const` and `formal` are leaves, and thus have no immediate descendants. The immediate descendant of a `un_op` node is its `left` attribute. The immediate descendants of a `bin_op` node are its `left` and `right` attributes. This is reflected in the following overloaded definition:

```
Define IDesc(n:const) = Empty;
Define IDesc(n:formal) = Empty;
Define IDesc(n:un_op) = n.left;
Define IDesc(n:bin_op) = n.left Union n.right;
```

In the following tree, the immediate descendants of a are <b, c> (where < > specifies an object collection). The immediate descendants of c are <d, e>. Nodes b, d, and e have no immediate descendants.



Next we will define the descendants of a node as the *reach* of its immediate descendants. The reach of a node will be defined as the union of the node itself with its descendants. For example, in the tree pictured above, the descendants of a are <b, c, d, e>. The reach of a includes the nodes <b, c, d, e, a>. To define descendants, we make use of the definition of reach, and to define reach, we make use of the definition of descendants. Such definitions are called *cyclic*. Under certain restrictions, such definitions are guaranteed to make sense. In the assertion language, they should be prefixed by the keyword Cyclic. The definitions would be expressed as follows:

```
Cyclic Define Desc(n:Exp) = Reach(IDesc(n));
Cyclic Define Reach(n:Exp) = n Union Desc(n);
```

Asserting that the structure is a tree may be expressed as three subsidiary assertions. First, we assert that the reach of the `left` and `right` attributes of a `bin_op` node do not intersect. This will guarantee that no cross branches exist.

```
Assert ForAll n In bin_op Do
       Reach(n.left) Intersect Reach(n.right) Same Empty Od;
```

In the example tree, nodes `a` and `c` are of type `bin_op`, since they have two children each. The reach of `a.left` is the reach of `b`, which is < >. The reach of `a.right` is the reach of `c`, which is <`c, d, e`>. Thus the assertion is true for node `a`. The reach of `c.left` and the reach of `c.right` are both < >, and so the assertion holds for `c` also.

Second, we assert that no node is a descendant of itself. This will guarantee the absence of cycles.

```
Assert ForAll n In Exp Do Not(n Sub Desc(n)) Od;
```

The reserved word **Sub** denotes the collection subset operator. A quick look at the example tree shows that this assertion is true for that tree.

Finally, we assert that all nodes are reachable from the root definition. This will guarantee that there are no disjoint components in the tree.

```
Assert Reach(Root.def) Same Exp;
```

The reach of the root of the example tree (`a`) was listed above. Indeed, it is the entire tree (i.e. all expressions).

The IDL translator will automatically check for cyclic definitions. Thus, if the Cyclic keyword is left out, the translator will put it in for you and apprise you of this. You should be aware when you write cyclic definitions, however, since they carry with them two important restrictions. First, they must return collections, not values. Second, they cannot contain an If clause within their bodies. These restrictions guarentee that the function is monotonic, which in turn guarantees that its evaluation will terminate [4]. The checker will let you know if you break either of these restrictions.

Cyclic definitions are useful, as above in asserting that a structure is a tree. However, their use is not always necessary. Most assertions you will probably wish to make should be realizable without resort to cyclic definitions.

4.4. Unix Commands

The following Unix commands will run the assertion checker on the assertions for the constant folding process. If the IDL specifications for the constant folding process are in a file called *constant_fold.idl*, the parse tree instance to be input is in a file called *in*, and the parse tree output by the process is in a file called *out*, the sequence of commands would be:

```
idlc constant_fold.idl -s symboltable
assertcodegen symboltable -o generated_code
idlcheck input:in output:out generated_code -e error_log
```

The list of errors, if any, will now be in the file *error_log*.

Alternatively, one could use Unix pipes and condense the three commands above into:

```
idlc constant_fold.idl -s - | assertcodegen |
        idlcheck input:in output:out -e error_log
```

CHAPTER 5

ADVANCED FEATURES

5.1. Private Definitions

Private definitions are those written by the user in a programming language such as C, Pascal, or Modula-2. They allow the user to express his definitions in a language other than the assertion language. A user may find this desirable if the assertion language does not prove suitable for a particular definition the user would like to make.

The body of the definition must be placed in a file that is identified to the assertion checker by using the `-P` option with *idlcheck.* Within the specification, a private definition is declared by stating a return type. When the checker sees such a definition, it will look in the indicated file for the definition's body, and execute that body. Control will then return to the checker.

5.2. Naming Assertions

It is possible to name assertions. For example:

```
Range_check Assert ForAll CF In federal_customer Do
                   CF.agency_code  <= 100 Od;
```

`Range_check` is the name of the assertion. The naming of assertions is necessary if one wants to make use of the **Without** statement in an IDL specification. For instance, if the output of a process has the same structure as the input, but certain assertions should no longer hold in the output, one could state in the output specification:

```
Without Range_check;
```

This assertion would then not be checked in the output structure, though all other assertions would be. In addition, naming assertions also improves communication between people.

Part III

Implementation

CHAPTER 6

Overview of Implementation

Implementation of the assertion checker is divided into three major phases. Each phase communicates with the others through IDL structures. Figure 1 displays the phases and their interactions. Ellipses denote programs and rectangles denote data structures.

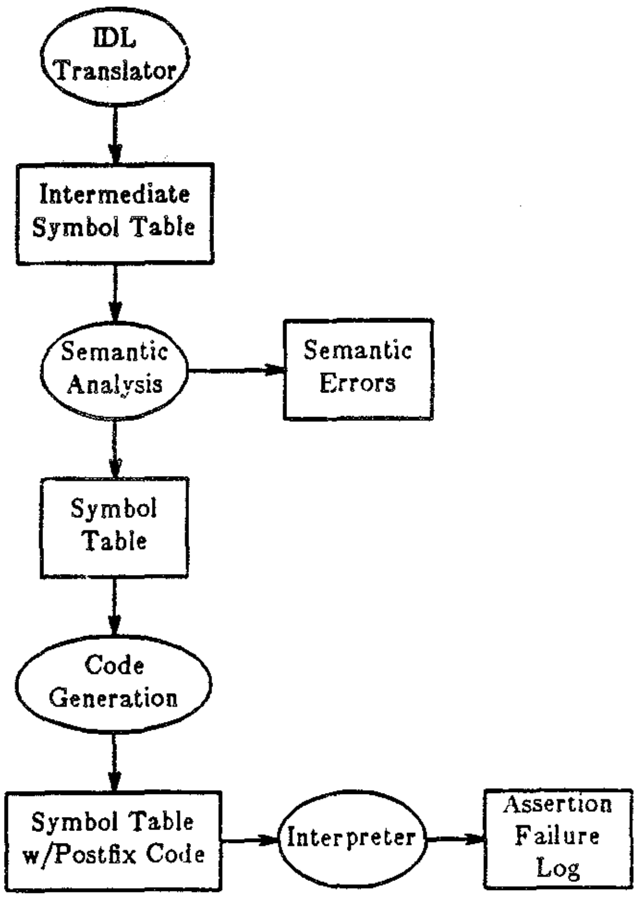


Figure 1

The front end and semantic analysis phases of the IDL translator deliver an intermediate symbol table to the semantic analysis phase of the checker. Assertions at this point are represented as abstract syntax trees. The assertion semantic analyzer outputs the symbol table. All information contained in the intermediate symbol table is preserved. The symbol table has added semantic information about assertions. Figure 2 (next page) displays detailed relationships between the several phases of the front end and semantic analysis. Each box represents a module in the entire process. Triangles at the edge of boxes represent IDL ports, where structures are read in or written out. Triangles pointing into a box represent readers, and triangles pointing out represent writers.

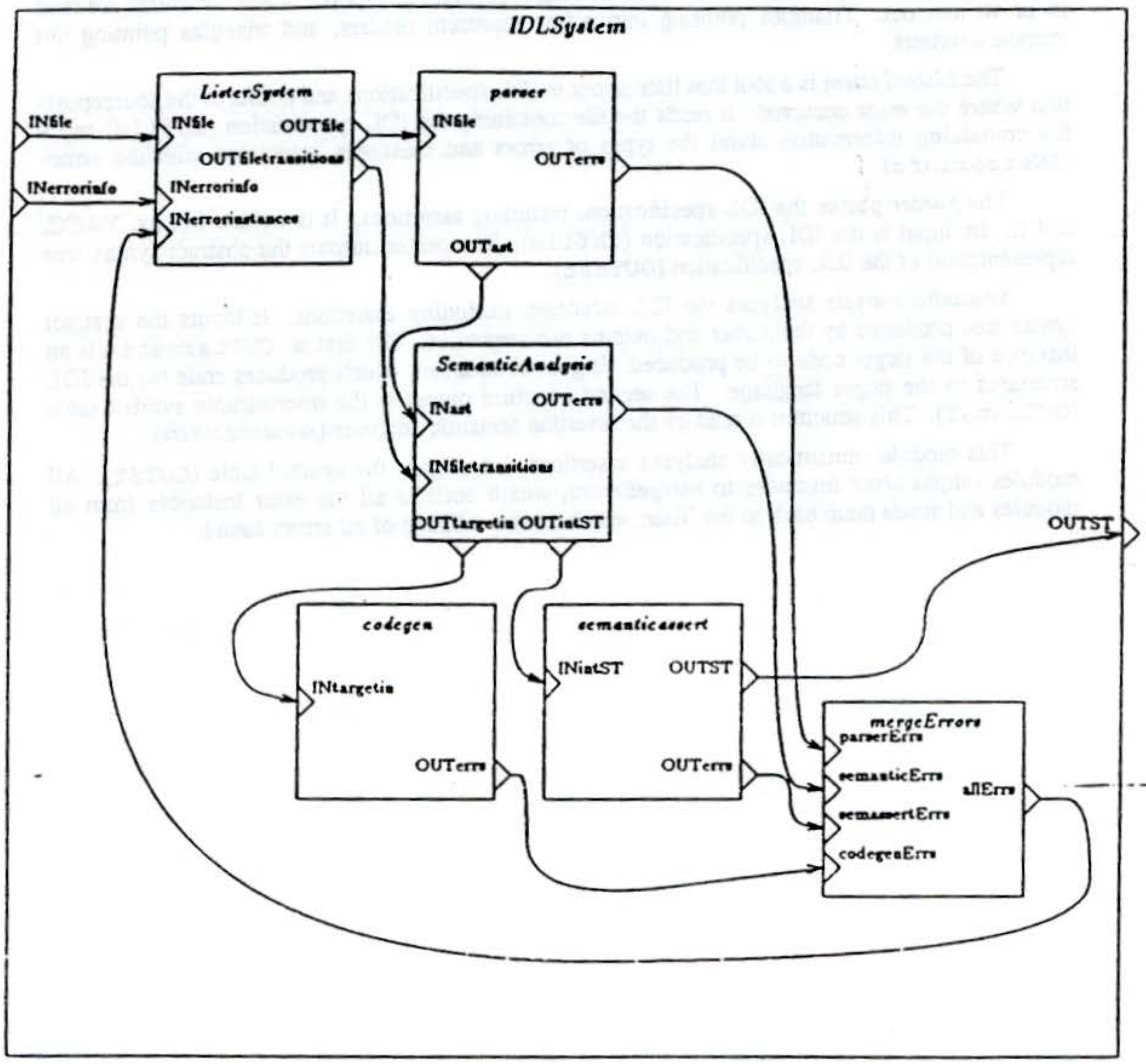
The *ListerSystem* is a tool that lists errors in IDL specifications and points to the sourceposition where the error occurred. It reads the file containing the IDL specification (`INfile`) and a file containing information about the types of errors and messages associated with the errors (`INerrorinfo`).

The *parser* parses the IDL specification, including assertions. It is written in Lex, YACC, and C. Its input is the IDL specification (`INfile`). The parser outputs the abstract syntax tree representation of the IDL specification (`OUTast`).

SemanticAnalysis analyzes the IDL structure, excluding assertions. It inputs the abstract syntax tree produced by the parser and outputs two structures. The first is `OUTtargetin` is an instance of the target code to be produced. It goes to *codegen*, which produces code for the IDL structures in the target language. The second structure output is the intermediate symbol table (`OUTintST`). This structure is read by the assertion semantic analyzer (*semanticassert*).

This module semantically analyzes assertions and outputs the symbol table (`OUTST`). All modules output error instances to *mergeErrors*, which collects all the error instances from all modules and sends them back to the lister, which creates a listing of all errors found.

Figure 2


IDLSystem
INfile
INerrorinfo
ListerSystem
INfile
OUTfile
OUTfiletransitions
INerrorinfo
INerrorinstances
parser
INfile
OUTerrs
OUTast
SemanticAnalysis
INast
OUTerrs
INfiletransitions
OUTtargetin
OUTintST
OUTST
codegen
INtargetin
OUTerrs
semanticassert
INintST
OUTST
OUTerrs
mergeErrors
parserErrs
semanticErrs
semassertErrs
codegenErrs
allErrs

Code generation creates a postfix code representation of the assertions that will be directly interpreted by the interpreter. The code is added to the symbol table, which is output to the interpreter.

Finally, the interpreter takes the information in the symbol table (which now includes postfix code as well as attributed tree representations of the assertions) and interprets each assertion. Output is an assertion failure log listing assertions that were not satisfied and information about values of intermediate expressions in unsatisfied assertions that might help the user discover why the assertion was not satisfied.

Figure 3 displays the relationships between the modules of the assertion checker. These modules include the assertion code generator, the interpreter, and the lister. The *CodeGenerator* reads the symbol table (`INST`) and generates the code (`OUTCode`) that the *Interpreter* executes. The interpreter inputs the generated code (`INCode`) and the structure instances and outputs a list of failed assertions (`OUTerrs`) with information about the failed assertion that can aid the user in discovering why the assertion failed. The *Lister* lists the failed assertions and associated messages in an assertion failure log.

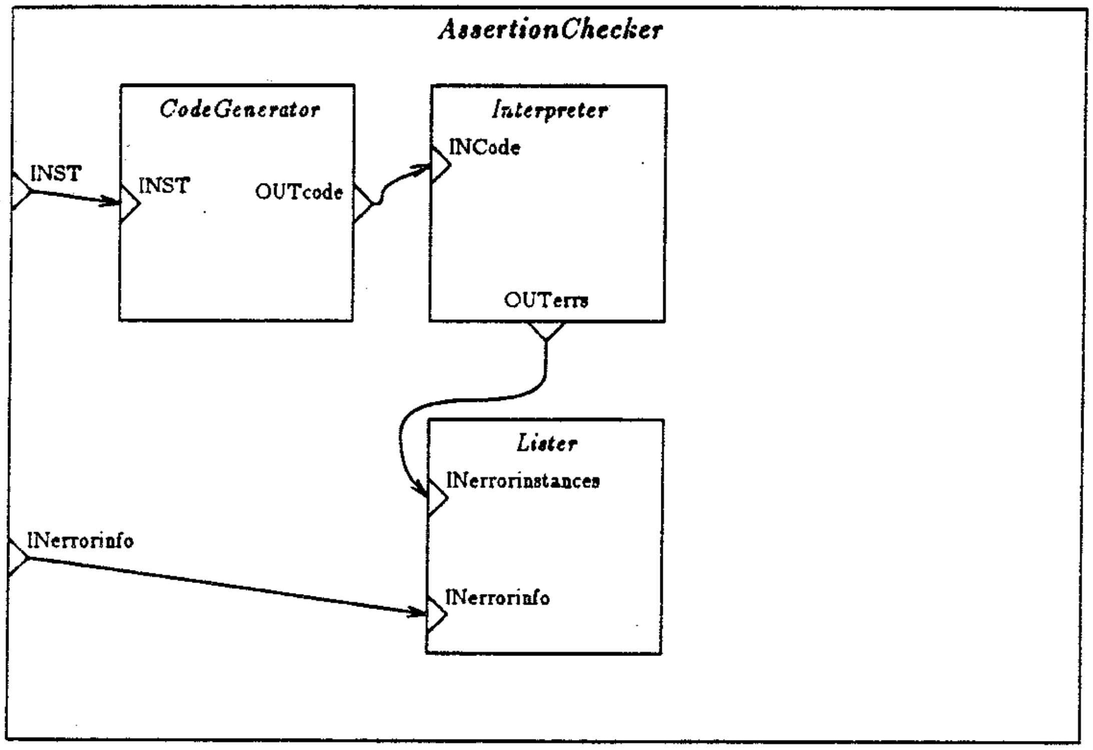


Figure 3

The following chapters describe in more detail the implementations of the three phases. Chapter 2 describes semantic analysis, chapter 3 describes code generation, and chapter 4 describes interpretation. Although assertion failure log generation is a part of the interpretation module, its description is given its own chapter.

CHAPTER 7

Semantic Analysis

7.1. Description

The assertion language is described by a strongly typed expression grammar. Semantic analysis involves type checking, name resolution, overloading analysis, and cyclic analysis. These will be discussed below.

Expressions in the assertion language may have types integer, rational, string, boolean, node, set or sequence of any of the above types, and object collection. A node type is a user defined type that is declared as an IDL node. In the case of object collections, semantic analysis must determine the possible types of the objects in the collection. There should be no runtime type errors. If the semantic analyzer cannot be certain that the types of the parts of an expression will make sense at runtime, then an error is returned. For example, the assertion

```
Assert ForAll c In Customer Do
      If Type(c) Same commercial_customer
      Then c.industry_code = 10
      Else True Fi
      Od;
```

would not be accepted. The control variable `c` iterates over all nodes of type `Customer`. Although the node `commercial_customer` has an attribute called `industry_code`, not *all* `Customer` nodes do. Therefore, the semantic analyzer cannot be certain that the control variable will have the named attribute. An object missing an attribute would cause a runtime error, and so the above assertion is semantically incorrect.

Control variables, formal arguments, type expressions with no port specified, and parameterless definition references cannot be distinguished by the parser. They are recognized only as names. Semantic analysis must resolve these names before typing them.

A quantifier is active if its body is currently being typed. Active quantifiers are kept on a stack. When a name is encountered, it is checked against the control names for all active quantifiers. If no match is found, the name is then checked against the names of all formal arguments in the active definition, if a definition body is currently being typed. (At most one definition is active at a time. The scope of formal names is local to the current definition.) If the name is still not bound, the names of all definitions that have no parameters are searched. If this fails, the name is assumed to be a type expression. The structure is searched for a node that has the same name as the name the analyzer is attempting to bind. If no node exists, the search for a binding is ended and a semantic error is reported.

Applications are examined to determine which definition they are referring to. The parser does not distinguish between applications of the language supplied functions (**Members**, **Head**, **Tail**, **Size**, and **Type**) and user defined functions. Semantic analysis makes this distinction, and in the case of user defined functions determines which definition (and if possible, which instance) the application refers to. Applications must also refer to a declared function, and the types of the actual arguments in the application must match the types of some instance of the function or the union of the instances.

For example, given the overloaded definition

```
Define CodeOf(c:commercial_customer) = c.industry_code;
Define CodeOf(c:state_customer) = c.state_code;
Define CodeOf(c:federal_customer) = c.agency_code;
```

the following assertions are both semantically correct:

```
Assert ForAll c In state_customer Do
       CodeOf(c) >= 1 Od;

Assert ForAll c In Customer Do
       CodeOf(c) >= 1 Od;
```

The first assertion is correct because the actual argument is of type `state_customer`, which is the type of the formal argument of one of the instances. The second assertion is also correct because, although there is no single instance whose formal argument type matches the actual argument's type (`Customer`), the union of the formal types (`commercial_customer`, `state_customer`, and `federal_customer`) includes the actual argument type. Here, the union of the instance formal types is exactly the actual argument type.

Overloading of user defined functions are checked. Instances of the same definition must be distinguishable by the number or types of their formal arguments. For example, the definition

```
Define CodeOf(c:Customer) = c.customer_number;
Define CodeOf(c:commercial_customer) = c.industry_code;
```

is semantically incorrect. The instances are not distinguishable because an argument of type `commercial_customer` matches both instances. At runtime it would not be known which instance to apply.

A *singleton* collection is a collection that is known at compile time to contain exactly one object. Singleton collections may act as values in expressions. The most common examples are dot expressions, which formally denote collections. The expression `c.customer_number` above means the value of the customer number, and can be used in arithmetic expressions. It is also correct to state an assertion about the object `c.customer_number`, that is, the collection containing the customer number (a collection containing one integer). For example, the expression `c.customer_number Union c.industry_code` results in a collection containing two integers (if the integers are distinct). The semantic analyzer must allow singleton collections as operands of value operators while disallowing other collections. It does this by examining the kind of operator to determine if a value or collection is required. If the operator requires a value, and an operand is a collection, then the collection is checked to see if it is a singleton collection. The following collections, and only the following, are singletons: formal parameter names, quantifier control variables, Root, Head, dot qualification of any of these forms, and If expressions where all expressions following Then and Else are one of these forms.

A *cyclic* definition is a definition that may call itself indirectly in the course of evaluating its body. Cyclic identical calls are permitted. A cyclic definition must return an object collection, but semantic analysis must determine the types of the objects in the collection. Since the body of a cyclic definition may include cyclic identical calls, care must be taken to ensure that typing stops. The collection of types is initialized to null (the empty set). The body of the definition is typed, with the set of types returned by any cyclic call assumed to be null. This process is then repeated with the set of types returned by a cyclic call set to the result of the last typing. Typing continues until the set of types returned does not grow. Since cyclic definitions are ensured to be monotonic (see [4], appendix A), this method will find all types returned. The current implementation does not handle cyclic definitions.

7.2. Algorithm

The semantic analyzer recursively descends the abstract syntax tree. Leaf expressions are typed prior to expressions closer to the root. The root (the assertion body) is last to be typed. The typing procedure is a long C switch statement. Each case handles a different kind of expression.

Noncyclic definitions are typed first. A directed call graph is created. Nodes are definitions and an arc from definition A to definition B indicates that definition A calls definition B. A depth first search is then performed on the call graph to determine if any of the noncyclic definitions are indeed in a cycle. If so, they are reclassified as cyclic. A reverse topological sort is then performed on the call graph. This provides the order in which the definitions must be typed. The definition typed first is that which calls no other definition. Since these are noncyclic definitions, the call graph is guaranteed to be noncyclic.

Cyclic definitions are then typed as explained above. (Cyclic type checking is not yet implemented.)

Each definition is then checked to ensure that all instances of the same definition return the same type. Finally, the bodies of the assertions themselves are typed in the order the assertions were written. The typing process is repeated for each structure and process in the IDL specification.

CHAPTER 8

Code Generation

The code generator produces postfix code semantically equivalent to the attributed expression tree representation of the assertion body. The postfix code is represented as an array of one byte integers. In the implementation language C, the code is an array of characters. Appendix C contains a detailed description of the code.

The expression tree is recursively traversed. Code is generated for a node's children before it is generated for the node itself. Most of the operation of the generator is straightforward. For example, the assertion

```
Assert Size(Root.list) = 2 And
       Head(Root.list) Sub Government_customer;
```

would result in the postfix code

Root	.list	Size	2	=	Root	.list	Head	Gov't_customer
1	2	3	4	5	6	7	8	9
Sub	And	EndAssert						
10	11	12						

The number underneath the instruction is the instruction's index in the code array. Each instruction is represented by one byte integers, except for those instructions that must reference an attribute. Such instructions cannot be coded by one byte. For examples, the dot operator has an attribute name, which can be any string. Type expressions have a type reference. Since the user may define his own types (node types), there may an indefinite number of these type references. Applications have a definition reference, and there may be an indefinite number of definitions. Integer, string, and rational tokens refer to far more possiblities than can be encoded in one byte. To deal with these attributes, five arrays are associated with each structure or process. The code generator places string names, types, definitions, integers, and rationals into these arrays. The generated code contains two-byte pointers into these arrays. This allows for 64K distinct objects of each type.

The actual code would be (the number in parentheses is the index of the instruction)

Instruction	Integer Code
(1) `Root`	8
(2) `.list`	63 1 0
(5) `Size`	55
(6) `2`	2 1 0
(9) `=`	32
(10) `Root`	8
(11) `.list`	63 1 0
(14) `Head`	52
(15) `Gov't_customer`	9 1 0
(18) `Sub`	14
(19) `And`	20
(20) `EndAssert`	66

The instructions with two extra bytes are those with attributes. `list` has an index into the string name array where the name "list" will be found. `Gov't_customer` has an index into the type array.

Interpretation of the code proceeds left to right. An evaluation stack keeps track of the interpretation. Some instructions (for example, Root, 2, Gov't_customer) result in a value being pushed on the evaluation stack. Other instructions pop the appropriate number of arguments and push the result. For example, `Size` pops one value off the evaluation stack and pushes the size of that value. `And` pops two values, and pushes `True` if the conjuction is true, `False` otherwise. All assertions are ended by the instruction `EndAssert`. This instruction pops the evaluation stack and returns the value, which is either `True` or `False`.

An If expression such as

```
Assert If Size(Root.list) = 3
     Then Head(Root.list).name = "ATT"
     Else True Fi;
```

would result in the code

Root	.list	3	=	JFALSE	23	Root	.list	Head	.name
1	2	3	4	5	6	7	8	9	10
"ATT"	=	JUMP	28	True	EndAssert				
11	12	13	14	15	16				

The instruction *jfalse* jumps to the location specified in the next instruction if the top of the evaluation stack is `False`. The instruction *jump* is an unqualified jump.

The actual byte codes would be (the number in parentheses is the index of the instruction)

Instruction	Integer Code
(1) `Root`	8
(2) `.list`	63 1 0
(5) 3	3 1 0
(8) =	32
(9) `jfalse`	50 23 0
(12) `Root`	8
(13) `.list`	63 1 0
(16) `Head`	52
(17) `.name`	63 2 0
(20) `"ATT"`	4 3 0
(23) =	26
(24) `jump`	48 28 0
(27) `True`	5
(28) `EndAssert`	66

The extra bytes after the jump instructions are absolute indexes in the code array where execution should continue.

The quantifier expression

```
Assert ForAll c In Customer Do
       c.customer_number >= 1 Od;
```

would result in the code

Customer	ForAll	c	.customer_number	1	>=	/
endForAll1						
1	2	3	4	5	6	7
5	EndAssert					
8	9					

The actual code for this assertion would be (the number in parentheses is the index of the instruction)

Instruction	Integer Code
(1) `Customer`	9 1 0
(4) `ForAll`	57
(5) `c`	61 1
(7) `.customer_number`	63 1 0
(10) `1`	1
(11) `>=`	31
(12) `endForAll`	59 5 0
(15) `EndAssert`	66

The two extra bytes after `endForAll` instruction represent an absolute index in the code array where the quantifier begins. The extra byte after the control instruction `c` is the control's nesting level, which is the nesting depth of the quantifier to which the control variable belongs. The `Customer` instruction will create the collection of all nodes in the structure of type `Customer`. `ForAll` sets up the execution of the quantifier body by placing the first object in the Customers collection where the control variable can get it. The instruction `c` takes one of the objects in the collection of Customers and pushes it onto the evaluation stack. The body of the quantifier is then executed. The `endForAll` instruction pops the evaluation stack. If the result is `False`, then the `ForAll` is false, and the interpretation of the `ForAll` is stopped. If the result is `True`, the current object in the Customers collection is removed, and control returns to the instruction indexed by the instruction after the `endForAll`.

Detailed explanations and examples of interpretation of code are provided in chapter 4 of this part. Special considerations in code generation will now be discussed.

Many instructions may take several different types of operands. For example, the equality operator may compare operands of type node, integer, rational, string, boolean, set, or sequence. The code generator examines the types of the operands of each operator and generates an instruction specific to those types of operands. For example, the expression `c.customer_number < 10` results in the instruction *num_less*, which denotes the less-than relation on numeric operands (integers or rationals). The expression `c.name = "ATT"` results in the instruction *str_less*, which denotes the less-than relation on string operands. Since the interpreter will not have to examine the types of the operands before operating, it is faster.

If an assertion is found to be not satisfied, the assertion checker will print out an assertion failure log containing information about the values of intermediate expressions leading to the unsatisfied assertion. The assertion failure log generator will need to walk down the expression tree (in a prefix manner), and yet have information about the result values of expressions which are interpreted in a postfix manner. To make this possible, the code generator saves in each node of the expression tree an index in the generated code array where that expression's code ends. It is at this point that the result of the expression is known. For more information about assertion failure log generation, see chapter 5 in this part.

Code generation is implemented with one recursive procedure. The procedure is one long switch statement. Each case handles one kind of expression. Code is generated for the operands of an operator before it is generated for the operator itself by recursively calling the code generation procedure for each operand. The result is postfix code.

CHAPTER 9

The Interpreter

The code generation phase of the assertion checker creates a sequence of interpreter instructions for each assertion and the body of each IDL definition instance. The interpreter can then be viewed as a virtual machine executing these instructions.

Instructions are represented by one byte integers. Attributes are represented as two additional bytes following the instruction they are associated with. These bytes are formed into a full size integer that serves as an index into an array containing pointers to the actual attributes. There are five of these *attribute arrays*, one each to contain integers, rationals, strings, type references, and definition references. The instruction sequence is represented as an array of bytes.

There are several auxillary data structures used by the interpreter: the *runtime stack*, the *quantifier collections list*, and the *control array*. We will describe each below.

The interpreter machine makes use of a runtime stack. Each instruction the machine encounters in the array results in some action involving the runtime stack and sometimes further actions involving a list of collections maintained in order to interpret quantifier instructions.

A quantifier specifies a collection of objects over which its control variable will iterate. The quantifier collections list is a sequence containing the collections for all active quantifiers. An active quantifier is one whose body is being executed. As quantifiers are encountered during interpretation, the iteration collections for the quantifiers are appended to the rear of the quantifier collections list. Collections toward the end of the list are more deeply nested than those toward the start. When a quantifier has been interpreted, it is no longer active and its iteration collection is removed from the quantifier collections list. If no quantifiers are currently being interpreted, the quantifier collections list will be empty. The collection associated with a quantifier which has just been interpreted is always the last collection in the quantifier collections list (at the rear), since the most recently completed quantifier is the most deeply nested.

A control array is used to implement the interpretation of control variables. The control array contains objects or collections, and is implemented as an array. When a control variable reference is encountered, the object or collection in the control array at the position pointed to by the control's level attribute is pushed onto the runtime stack. This allows a control reference to be replaced with its actual value in constant time. The control array is modified whenever a quantifier evaluation begins and ends. Formals use a control array similarly. The control array for formals is modified whenever an application evaluation begins and ends.

In order to clarify the operation of the interpreter, we will trace the contents of the runtime stack and control array as two different assertions are interpreted.

The first example is the assertion

```
Assert Size(Root.list) = 2 And
      Head(Root.list) Sub Govenment_customer;
```

The code generated for this assertion is

Root	.list	Size	2	=	Root	.list	Head	Gov't_customer
1	2	3	4	5	6	7	8	9
Sub	And	EndAssert						
10	11	12						

The string reference array contains the string "list". The type reference array contains the type Government_customer. Initially, the runtime stack and control array are empty. Figure 4 shows the state of the runtime stack after the execution of each operation. Objects are identified by their labels, which are identifiers beginning with the character 'L' and followed by one or more digits. The control array is not used in interpreting this assertion (there are no quantifiers), so it is not shown.

The execution proceeds as follows (TOS refers to Top-Of-Stack and the number in parentheses indicates the order in which the instruction is executed):

(1) Root	Push the root object of the structure.
(2) .list	Pop the stack. Push the list attribute of TOS.
(3) Size	Pop the stack. Push the size of TOS.
(4) 2	Push the integer 2.
(5) =	Pop the stack twice. Push FALSE since 2 ≠ 3.
(6) Root	Push the root object of the structure.
(7) .list	Pop the stack. Push the list attribute of TOS.
(8) Head	Pop the stack. Push the head of TOS.
(9) Gov't_ customer	Push the collection of all Gov't_customer nodes.
(10) Sub	Pop the stack twice. Push FALSE since {L2} is not a subset of {L3 L4}.
(11) And	Pop the stack twice. Push FALSE since both values popped are false.

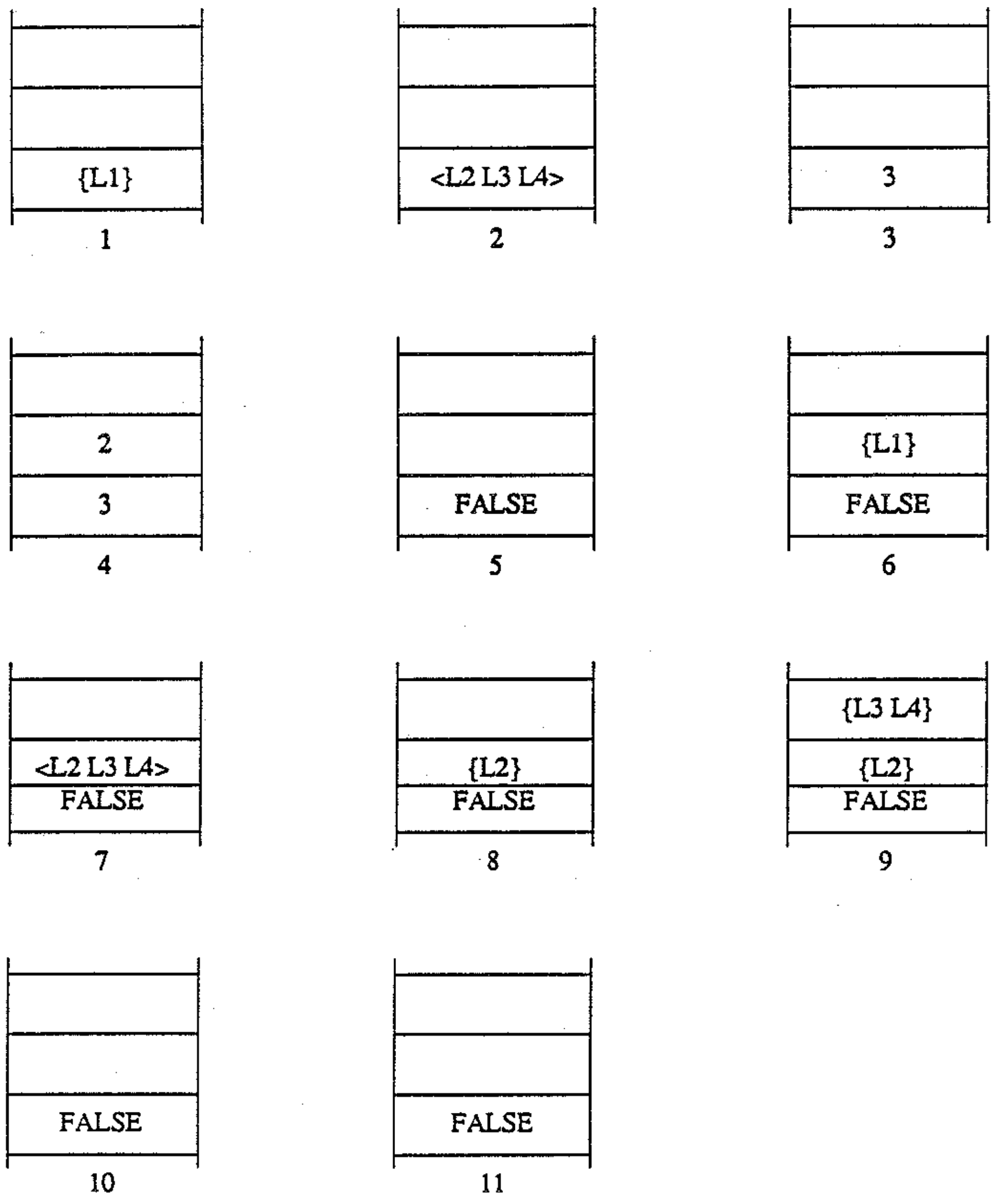

{L1}
1
<L2 L3 L4>
2
3
3
2
3
4
FALSE
5
{L1}
FALSE
6
<L2 L3 L4>
FALSE
7
{L2}
FALSE
8
{L3 L4}
{L2}
FALSE
9
FALSE
10
FALSE
11

Figure 4

The second example is the following assertion that contains a quantifier:

```
Assert ForAll C In Customer Do
       C.customer_number < 2 Od;
```

The code generated for this assertion is:

Customer	ForAll	C	customer_number	2	<	endForAll	EndAssert
1	2	3	4	5	6	7	8

Figure 5 displays the state of the runtime stack and control array after the execution of each instruction. The quantifier collections list, while important, is not shown. The execution proceeds as follows:

(1) `Customer`	Push the collection of all Customer nodes.
(2) `ForAll`	Pop the runtime stack. Remove an object from TOS and place it in control array at level 1. Place modified TOS on end of quantifier collections list.
(3) `C`	Push the object found in the control array at level 1.
(4) `customer_ number`	Pop the runtime stack. Push the customer_number attribute of TOS onto the runtime stack.
(5) 2	Push the integer 2 onto the runtime stack.
(6) <	Pop the runtime stack twice. Push TRUE onto the runtime stack since 1 < 2.
(7) `endForAll`	Pop the runtime stack. Since TOS is true, remove another object from the Customers collection, found on the end of the quantifier collections list, and replace the object in the control array at level 1 with the new object. Resume control at instruction 3.
(3) `C`	Push the object found in the control array at level 1.
(4) `customer_ number`	Pop the runtime stack. Push the customer_number attribute of TOS onto the runtime stack.
(5) 2	Push the integer 2 onto the runtime stack.
(6) <	Pop the runtime stack twice. Push FALSE onto the runtime stack since 2 is not less than 2.
(7) `endForAll`	Pop the runtime stack. Since TOS is false, push FALSE onto the runtime stack. The quantifier is not satisfied.
(8) `EndAssert`	Pop the stack. Return TOS, which is `False` as the result of the interpretation. End the assertion interpretation.

The object for which the quantifier failed is still in the control array. This is useful for production of the assertion failure log, as will be discussed in chapter 5 of this part.

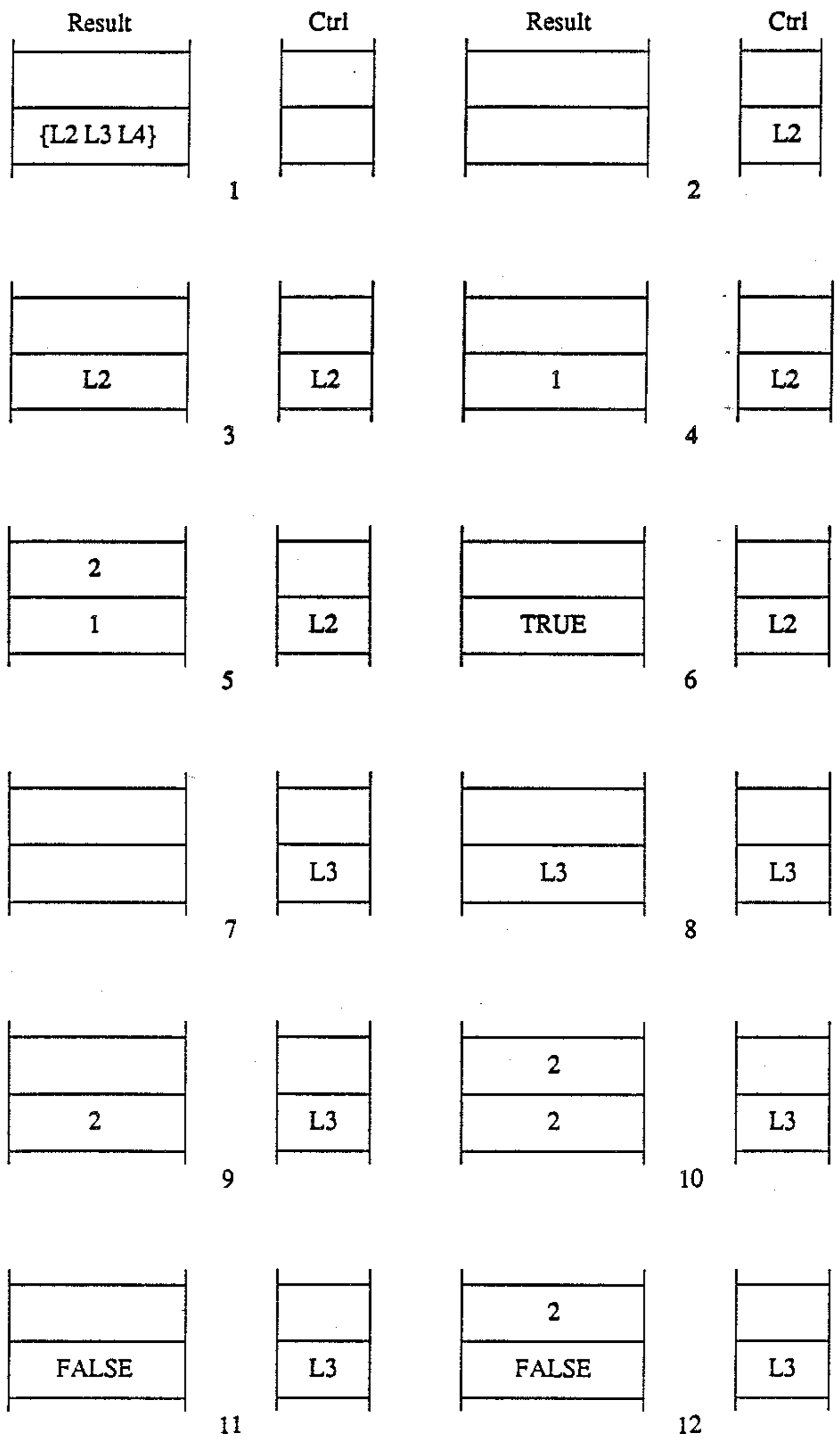

Result
Ctrl
Result
Ctrl
{L2 L3 L4}
1
L2
2
L2
L2
3
1
L2
4
2
1
L2
5
TRUE
L2
6
L3
7
L3
L3
8
2
L3
9
2
2
L3
10
FALSE
L3
11
2
FALSE
L3
12

Figure 5

The interpreter consists of one long switch statement. Each case executes a different instruction.

CHAPTER 10

Assertion Failure Log Generation

When the assertion checker finds an assertion that is not satisfied, it prints an assertion failure log containing information about the values of the intermediate expressions leading to the unsatisfied assertion. The assertion failure log should contain information that could be useful to the user in discovering why the assertion was not satisfied, yet not contain so much irrelevant information that the useful information is hidden.

The assertion failure log is generated through a preorder traversal of the expression tree. The traversal depends on the availability of certain information saved during code generation and interpretation.

During code generation, each subexpression is provided a pointer to the position in the code array where that expression's code ends. It is at this position that the interpreter determines the result of that subexpression, since the code is postfix. Whenever the interpreter stacks a result on the run stack, it also saves this result in a result array. This array is the same size as the code array, but holds run stack entries, not instructions. The result is stored in the result array at the same index as the instruction that produced the result is stored in the code array. Thus, a subexpression in the tree points not only to the end of its code in the code array, but also to the position in the result array where the value of its result is stored.

As the assertion failure log routine traverses the expression tree, it decides whether to print information about each subexpression (and by extension, subexpressions of that subexpression). Information about a subexpression is printed if that subexpression was not satisfied, unless it is part of a negation (Not), in which case the subexpression is printed if it is satisfied. Determining whether a subexpression was satisfied involves looking in the result array at the position saved during code generation.

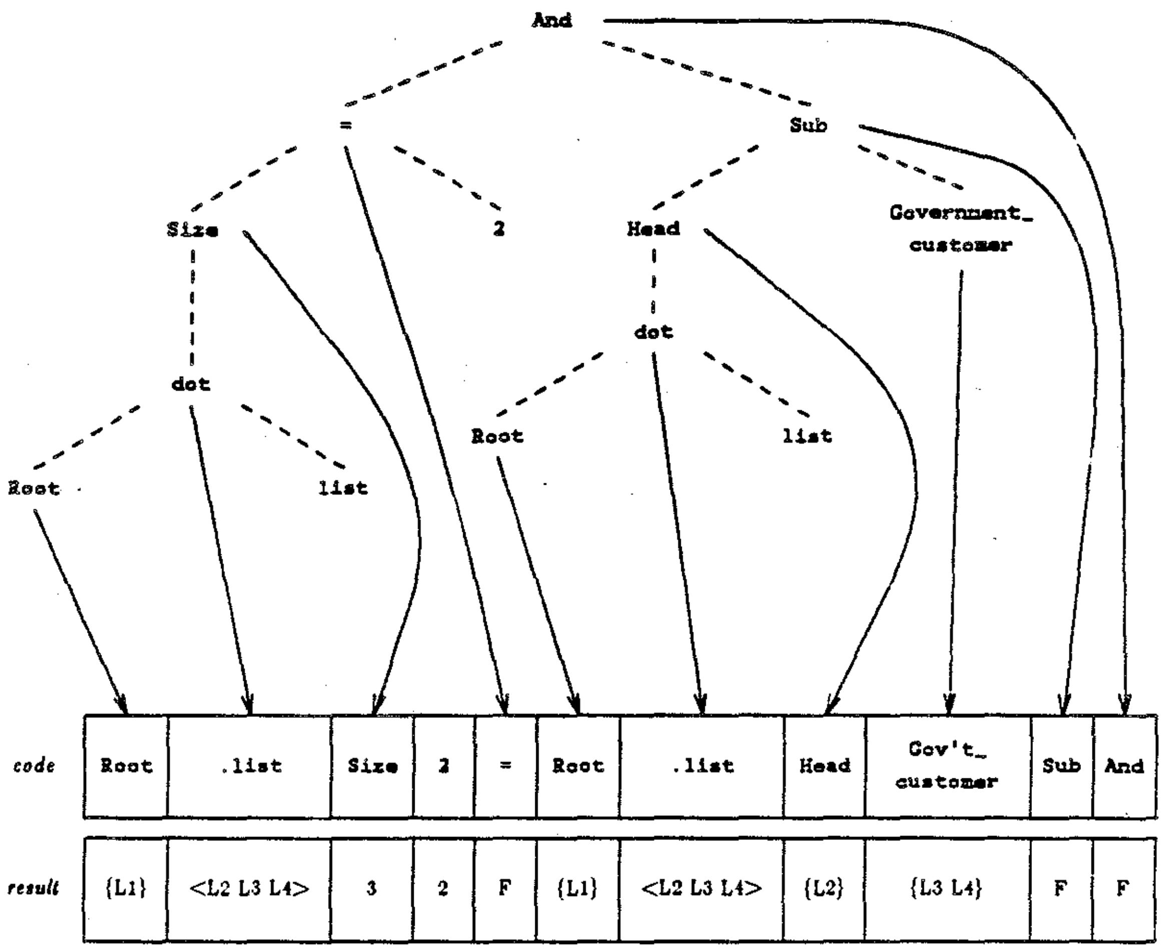
Results of non-boolean subexpressions are also saved in the result array. Information about these subexpressions is printed if the subexpression is part of an unsatisfied expression (or satisfied if part of a negation).

Objects are represented by their labels, which are strings beginning with the character `L`, followed by one or more digits. Sets and collections are enclosed in curly brackets ({ }). Sequences are enclosed in pointed brackets (< >).

Figure 6 illustrates the relationships between the various data structures for the example assertion:

```
Assert Size(Root.list) = 2 And
       Head(Root.list) Sub Government_customer Od;
```

Dotted lines denote descendant node relationships in the expression tree, while solid lines denote that the expression contains the index in the code array of the instruction pointed at.



code	Root	.list	Size	2	=	Root	.list	Head	Gov't_ customer	Sub	And
result	{L1}	<L2 L3 L4>	3	2	F	{L1}	<L2 L3 L4>	{L2}	{L3 L4}	F	F

Figure 6

The pointer from the integer literal 2 is not shown because it is not used by the log generation routine. There is no need to print the result of a literal expression (excepting Root).

To improve readability, the error message is indented. Information about an expression is indented by a number of levels equal to the depth of the expression in the tree. In addition, the value of an expression is preceded by the symbol > (because) if the expression is farthest left at its level, and by the symbol & (and) if the expression follows another expression at the same level. The output produced for the above assertion, assuming that the assertion was not satisified and that both operands of the conjunction were false, is:

```
Assertion is false.
> And not satisfied.
    > Equality not satisfied.
        > Size is 3
            > .list is { < L2 L3 L4 > }
                > Root is { L1 }
    & Subset not satisfied.
        > Head is { L2 }
            > .list is { < L2 L3 L4 > }
                > Root is { L1 }
        & Government_customer is { L4 L3 }
```

Quantifier operators (**ForAll** and **Exists**) are handled a little differently. The quantifier expression contains the index of the end of quantifier instruction. The result of the quantifier execution is placed in the result array at this index by the end of quantifier instruction. The end of quantifier instruction also places the object that caused the quantifier to be false (in the case of a ForAll) or to be true (in the case of an Exists) in the result array at the succeeding index.

Figure 7 illustrates these relationships for the assertion:

```
      Assert ForAll C In Customer Do
            C.customer_number <= 2 Od;
```

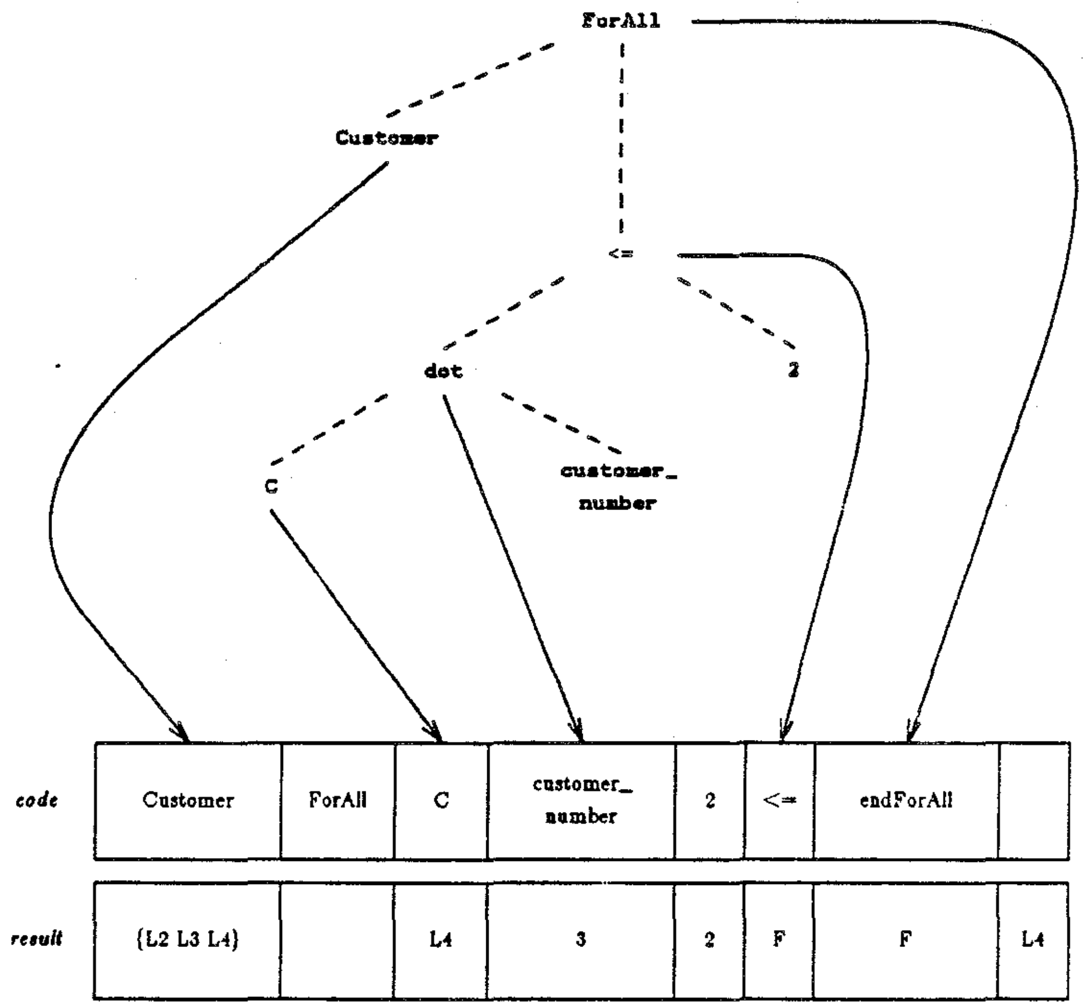


Figure 7

CHAPTER 11

Recommended Improvements to the Current Implementation

This chapter describes several recommendations for improving the current implementation. These involve either implementing features of the language that are not yet handled, or methods to increase the speed of execution of the interpreter.

The current implementation includes all features of the IDL assertion language as described in [3] excepting cyclic and private definitions.

Cyclic functions and private definitions should be implemented. They are part of the language definition, and add expressive power to the language. Of the two, the implementation of cyclic functions is more important. It also appears to be fairly involved.

As now implemented, the interpreter evaluates the body of a definition each time it is called. The result of the call is placed on the runtime stack, but not otherwise saved. Thus, the result value will be lost after further interpretation, and the next time the definition is called, the same evaluation will take place. This re-evaluation is wasteful, especially if the evalution requires a time consuming walk through an entire IDL structure. A straightforward attempt at a solution to this problem is to cache results returned by definition body evaluations. Performance analysis should be done to determine the optimum size of the cache.

It is our feeling that caching would dramatically improve the performance of the checker on many assertions and would not be difficult to implement.

Implementation of the Type function and type expressions is slow. To collect all objects in an IDL structure with a given type, the entire structure is traversed. Each object is encountered and its type compared to the given type. If matched, the object is placed in the collection. Slow implementation of Type and type expressions is especially a problem since both Type and type expressions are expected to be widely used.

Lamb [4] suggests linking all objects of the same type as the objects are read in by the IDL reader. Then implementation of Type and type expressions would involve finding one object of the correct type and then following pointers. This would speed up interpretation at the cost of slowing down input of structure instances. However, the speed gained in interpretation may be more than that lost in reading.

Increasing the speed of the implementation of type expressions is very important. Type expressions are heavily used and can involve hundreds of objects. Modifying the reader, however, may be difficult.

If the suggestion to include the abbreviation for `Type(x) Sub Y` in the language is accepted, then the most common use of the Type function would be eliminated, and the slow implementation of Type would be less of a problem.

Binary operators acting on integers and rationals should be subdivided into more instructions. In the present implementation, relational opeators such as `greater than` are translated into different instructions by the code generator if the types of the operands are string as opposed to numeric. All numeric operands map to the same instruction. We recommend new instructions that depend on whether the operands are integer, rational, or both. For each of the relational and arithmetic operators, the following instructions should be added: `both_integer`, `both_rational`, `integerLeft_rationalRight`, and `integerRight_ rationalLeft`. This improvement could be implemented simply and quickly.

The assertion failure log provides information to aid the user in discovering why a particular assertion was not satisfied. The algorithm that generates the failure log suppresses some information that would not be useful. For instance, if a conjunction is not satisfied, information regarding only the operand that was not satisfied is provided. However, redundant information may still be reported. For example, if a quantifier is not satisfied, the log reports for which object the quantifier is not satisfied, and then reports that object again when providing information on the body of the quantifier. An improved algorithm would report the object only once.

Suppressing redundant information is a convenience. An algorithm that suppresses all irrelevant information might not be worth the trouble. Some middle ground would probably be most practical.

The assertion failure log denotes a collection by listing the labels of the objects in that collection. Collections may be very large. Currently, the log stops listing objects after fifteen have been printed out, and then prints three periods (. . .) if there are more. An improved version would allow the user to see the remaining objects if desired. This facility is fairly important, and should be given a higher priority than the suppression of redundant information described above.

Part IV

Conclusions

CHAPTER 12

Conclusion

Our thesis is that a useful IDL assertion checker tool can be implemented. This document describes the implementation of the checker tool. The existence of a functional implementation serves to prove our thesis.

We have also demonstrated that the assertion language can be strongly typed at compile time. That is, it can be guaranteed that there will be no runtime type errors when running the checker tool. Further, strong typing does not limit the expressive power of the assertion language.

In addition, we have shown that useful information can be automatically provided by the interpreter. This information can aid the user in discovering why a particular assertion was not satisfied.

Although not ignored, efficiency was not of primary interest in the current implementation. We have discussed improvements to the current implementation that would improve performance.

Finally, we believe that the checker can be a useful tool in aiding the debugging of large systems programs developed with the Interface Description Language.

CHAPTER 13

Future Research

This chapter presents several recommendations for future research and improving the assertion checker. The recommendations are split into two groups: those that are extensions to the language and those that are changes to the semantics of the language.

13.1. Language

The form

```
Type(x) Sub Y
```

is expected to occur often. The formal meaning of the above construct is to assert that the collection containing all objects of the same type as `x` is a subset of the collection containing all objects of the same type as `Y`. The object `x` is usually a control variable or formal argument, and the object `Y` is usually a type expression. A user would most likely wish to assert that the type of the object `x` *is* `Y`, but the language forces the above expression. A more natual way of expressing the meaning of this construct would be:

```
x Is Y
```

We suggest that this abbreviation be adopted.

Lamb points out [4] that there is no way in the assertion language to create a subset of a collection that contains all objects in the collection possessing a certain property. He proposes a construct to produce such a subset. We agree with the need for the construct, but propose a simpler syntax. The new construct is a variant of the **Forall** operator. Like **ForAll**, it iterates over a collection. Unlike **ForAll**, it returns a collection rather than a boolean value. The proposed construct is:

```
Select x In <exp> Do <body> Od
```

<exp> must return a collection. <body> must return a boolean. The **Select** returns the collection containing all objects in <exp> that make <body> true.

Conditional execution of assertion expressions, dependent on the type of an object, is expected to be widely used. Because the assertion language is strongly typed, the use of the If expression is not always sufficient to express simple assertions a user might wish to make. For example, given the IDL declaration

```
exp ::= leaf | inner ;
leaf => depth : Integer;
inner => left : exp, right : exp;
```

the definition of the immediate descendants of a node of type *exp* would have to be overloaded. The definition

```
Define IDesc(e:exp) = If Type(e) Sub leaf Then Empty
                                          Else e.left Union e.right Fi;
```

is natural but semantically incorrect, since e might be a *leaf*, which has no *left* or *right* attributes. We propose adding the following Case construct:

```
Case <name> Is <exp0>
<type1> Do <exp1> Od
<type2> Do <exp2> Od
        .
        .
        .
{Otherwise Do <expn> Od}?
End
```

<name> is bound to *<exp0>*, and is local to the Case statement. If the type of *<name>* is *<type1>*, then execute *<exp1>*. Inside *<exp1>*, *<name>* is typed to *<type1>*. If the type of *<name>* is *<type2>*, then execute *<exp2>*. Inside *<exp2>*, *<name>* is typed to *<type2>*. Otherwise, execute *<expn>*.

Each *<type>* must be a type that *<name>* can have. If **Otherwise** is missing, then all types that *<name>* can have must be specified in the *<type>* list. *<name>* cannot appear in the Otherwise expression, *<expn>*. The types of all the *<exp>*s, other than *<exp0>*, must be the same. This type is the type of the entire Case expression.

With the new construct, the definition could be

```
Define IDesc(e:exp) = Case x Is e
        leaf Do Empty Od
        inner Do x.left Union x.right Od
End;
```

Similarly, quantifier assertions may be made in a more natural way.

No expressive power is added to the language. The proposed change makes the language more natural and convenient.

13.2. Semantics

The semantics of the dot operator are not clean. The problem arises in dealing with attributes of a class as opposed to attributes of the specific members of the class. For example, consider the following IDL specification:

```
A ::= B | C ;
B => x : Integer ;
C => x : String ;
```

The meaning of expression `A.x` is not clear. Is it a collection of **Integer** or of **String**? Right now, the language allows `A.x`, since every member of `A` has an attribute `x`. The result is a collection containing objects of both types. Both these types must satisfy any further type checking the dot expression is involved in. We recommend making `A.x` a semantic error, since the class `A` does not explicitly have an attribute `x`. `A.z` would be semantically correct only if an explicit attribute declaration existed:

```
A => z : Boolean ;
```

Here, the result of `A.z` is a collection of type **Boolean**.

Implementation of semantic checking of dot expressions would be simplified if the above semantics were adopted. It would no longer be necessary to check all descendants of a class for the given attribute. Subsequent type checking involving the dot expression would also be simplified, since the resulting collection of a dot expression would contain objects of at most one type.

This recommendation was proposed by John Nestor.

The semantics of the Type function in process assertions is not clear. In a structure assertion, Type returns the collection of all objects within the structure that are of the same type as the objects in the function argument. In a process assertion, it cannot be determined which structure (or all structures) is meant. We feel that the use of Type in a process assertion never has the same meaning as in a structure. Rather, a user would intend to assert that a particular object is a certain type. We therefore propose that Type be allowed only within a structure assertion, and that the `x Is Y` construct be used in process assertions.

Part V

Appendices

Appendix A

Formal Definition of the Assertion Language

1. Introduction

This document includes the formal syntax of the IDL assertion sub-language and the informal semantics of that language. The grammar is from [3], and is in extended BNF.

IDL data structures are set up as graphs. The nodes of the graphs contain attributes whose values are integers, rationals, booleans, strings, nodes, sequences, or sets. Assertions provide a means to express restrictions on the structure of the IDL graph itself, the values of specified attributes within the graph, or to relate conditions in a structure read on an input port to conditions in a structure written on an output port.

The syntax of assertions is as follows:

```
<assertion>      ::=   <assert stmt> | <definition>
<assert stmt>    ::=   { <name> }? Assert  <expression>
```

A definition is a user created entity which returns some value. Definitions are discussed in Section 3. The optional assertion name is used to identify the assertion for humans and to permit the appearance of the assertion in a Without clause. The expression must be of type boolean, since assertions are either true or false.

2. Expressions

Expressions form a conventional expression grammar with operator precedence levels. Or and Union have the lowest precedence and "*" and "/" have the highest precedence. The syntax is as follows:

```
<expression>     ::=   <1exp> | <expression> <1op> <1exp>
<1op>            ::=   Or  |  Union
<1exp>           ::=   <2exp> | <1exp> <2op> <2exp>
<2op>            ::=   And  |  Intersect
<2exp>           ::=   { Not }? <3exp>
<3exp>           ::=   <4exp> | <3exp> <3op> <4exp>
<3op>            ::=   =  |  ~=  |  <  |  <=  |  >  |  >=  |  In  |
                       Same  |  Psub  |  Sub
<4exp>           ::=   { <4op> }? <5exp> | <4exp> <4op> <5exp>
<4op>            ::=   +  |  -
<5exp>           ::=   <primary exp> | <5exp> <5op> <primary exp>
<5op>            ::=   *  |  /
<primary exp>    ::=   { <name> : }? <type>
                       | <literal>
                       | ( <expression> )
                       | <primary exp> . <name>
                       | <name> ( <actuals> )
                       | <if expression>
                       | <quantified expression>
<literal>        ::=   True  |  False
                       | { <name> : }?  Root
                       |  Empty
                       | <integer>  | <rational>  | <string>
<actuals>        ::=   <expression> { , <expression> }*
```

There are two broad kinds of types an expression may have. IDL types include integer, boolean, rational, string, set, sequence, and node. The other kind of type is an object collection type.

A set is an IDL object which consists of an unordered group of IDL objects of some type. A sequence is an ordered list of IDL objects of some type. The component type of a set or sequence cannot be a set or sequence. An object collection is produced by assertions. It consists of objects of any IDL type. Its properties are similar to the IDL set type, but operations associated with the object collection type are distinct from those associated with the IDL set type.

2.1. Literals

boolean	the values `True` or `False`
integer	the standard integer type; the syntax is identical to that in the ASCII External Representation Language [3].
string	sequence of ASCII characters between quotes. Any ASCII character may be represented, including printable characters, blanks, and non-printable control characters; the syntax is identical to that in the ASCII External Representation Language [3].
`Empty`	denotes the empty object collection
`Root`	denotes the collection containing only the root node object of the structure in which the expression appears.
<type>	denotes the collection of all objects of the specified type in the structure in which the expression appears. This expression cannot appear in a process assertion.
<name>	where name is a quantifier name (see Section 2.7). Denotes the object currently assigned to the quantifier.

2.2. Infix Operations On Values

Operations on values of IDL types are straightforward. They include:

* boolean: `=, ~=, And, Or, Not`
* integer & rational: `=, ~=, <, <=, >, >=, +, -, *, /`
* string: `=, ~=, <, <=, >, >=`
* set: `=, ~=, In`
* sequence: `=, ~=, In`
* collection: `In`
* node: `=, ~=`

Most of these operations should be clear. Two booleans, integers, rationals, or strings are equal if they have the same value. Two sets are equal if they contain the same objects. Two sequences are equal if they contain the same objects in the same order. Two nodes are equal if they are the same object. A string is less than a second string if it is lexicographically less than the second string, using the lexicographic ordering defined by the ASCII character set. An object is In a set or sequence if it is an element of that set or sequence. An object is In a collection if it is contained in the collection.

2.3. Infix Operations On Object Collections

Typically, the system designer will create definitions (described in Section 3) using object collection type operations to create a collection of objects, and then make some assertion about the collection.

The following forms specify object collections:

<portname> : <type>	denotes the collection of all objects in a structure of the specified type. The expression must appear in a process declaration and refers to the structure associated with the named port.
<portname> : `Root`	denotes a collection containing only the root node object of a structure. The expression must appear in a process declaration and refers to the structure associated with the named port.
`Union`	object collection union. The operands must contain objects of the same type.
`Intersect`	object collection intersection. The operands must contain objects of the same type.
<primary exp> . <name>	of an object collection containing only node objects returns an object collection containing all objects associated with the specified attribute of all nodes in the original object collection. Each node object in <primary exp> must have <name> declared explicitly as an attribute.
`Same`	two object collections are the same if and only if they contain exactly the same objects. The assertion language forces one to distinguish between object collections and IDL types. If *a* and *b* are object collections, then the form *a* = *b* is semantically incorrect.
`Sub`	object collection subset.
`Psub`	object collection proper subset.

2.4. Prefix Operations

<name> (<actuals>)	denotes an application of a user defined definition (see Section 3). <name> is the name of the definition to apply. <actuals> is an ordered list of arguments to the defintion.

The following are supplied functions:

`Head ( `*seq*` )`	*seq* must be of type sequence. `Head` returns the first object in *seq*.

`Members(` *setorseq* `)`	*setorseq* must be of type set or sequence. `Members` returns the object collection containing the objects in the set or sequence *setorseq*.
`Size(` *arg* `)`	*arg* may be of type string, set, sequence, or collection. If a string, `Size` returns the length of the string. If a set, sequence or arbitrary collection, `Size` returns the number of objects in the set, sequence or collection. If a singleton collection (a collection that must contain exactly one object), then `Size` returns the size of the object contained in *arg*, which must be a string, set, or sequence.
`Tail(` *seqv* `)`	*seqv* must be of type sequence. `Tail` returns the sequence obtained by removing the first object in *seqv*.
`Type(` *n* `)`	*n* must be an object collection. If in a structure, `Type` returns the object collection containing all objects in that structure with the same type as those in the object collection *n*. If in a process, `Type` returns the type of the object(s) in the object collection *n*.

2.5. Object Collections as Values

Value operations may operate on object collections if it can be determined at compile time that the collection will consist of exactly one object. The value of a singleton collection is the value of the object in the collection. Only the following object collections may be used as operands of value operations: a quantifier name (see Section 2.7), a formal parameter within a definition body, the {<name>:}? `Root` form, the `Head` form, the dot qualification of one of these forms, and `If` expressions where all expressions following `Then` and `Else` are one of these forms.

2.6. If Expressions

An `If` expression has the syntax:

```
<if exp> ::=  If <expression> Then <expression>
                { OrIf <expression> Then <expression> }*
                Else <expression>  Fi
```

The `OrIf` clause is semantically equivalent to an `Else If`. Syntactically, though, the `OrIf` clause does not need to be closed by a `Fi` while each `Else If` clause does. The expressions following `If` and `OrIf` must be of type boolean. The expressions following `Then` and `Else` must be of the same type.

2.7. Quantifiers

Quantified expressions have the syntax:

```
<quantified exp> ::= { ForAll  |  Exists } <name>
                          In <expression> Do <expression>  Od
```

The expression following `In` must be an object collection type. The <name> is an iterator which is assigned individual objects from the collection returned by the <expression> following `In` and may be used only within the `Do - Od` expression. Despite the `Do`, the expression between `Do` and `Od` must be of type boolean. Nothing is "done". It simply expresses a property which all objects (the `ForAll` variant) or at least one object (the `Exists` variant) in an object collection must have. The expression following `In` must be an object collection type. The expression between `Do` and `Od` must be of type boolean.

3. Definitions

Definitions are used by the designer to create collections of objects about which assertions may be made. The syntax is:

```
<definition>          ::=  <privateDefinition> | <IDLdefinition>
<privateDefinition>   ::=  Define <name> {<formals>}?
                           Returns <type>
<IDLdefinition>       ::=  { Cyclic }?  Define <name>
                           {<formals>}? = <expression>
<formals>             ::=  ( <formal> {, <formal>}* )
<formal>              ::=  <name> : <type>
```

There are three kinds of definitions.

A *private definition* specifies a *return* type and its body must be linked with the assertion checker. This allows the body to be expressed directly in a target programming language, such as C.

Non-cyclic definitions, in which the keyword `Cyclic` is omitted, return values or object collections. Recursion is permitted, but the definition may not be cyclic, i.e. its evaluation must not involve a recursive call with the same arguments as on a previous call, termed a *cyclic identical call.*

Cyclic definitions, in which the keyword `Cyclic` is specified, return object collections. A cyclic definition may call itself indirectly in the course of evaluating its body. Cyclic identical calls are permitted. The result of a cyclic definition call is the minimum fixed point solution [4]. The body of a cyclic definition may not include an `If` expression. This restriction ensures monotonicity and thus the existence of a minimum fixed point result.

A definition need not have any parameters.

Overloading of definitions, in which the meaning of the definition depends on the types of its arguments, is allowed, provided the different versions of the definition can be distinguished by formal parameters. Two definition instances are distinguishable by their formal parameters if there is no set of arguments whose types match the formal parameter types of both instances. Instances of the same definition must return the same type.

Definitions are invoked with the <name> `(<actuals>)` form of <primary exp> (see Section 2.4). The type of each actual expression must match the specified type of the corresponding formal parameter of exactly one instance of the declared definition.

The type of a definition is the type following `Returns` if the definition is private, or the type of the expression, the definition body, following the "=" sign if the definition is an IDL definition.

4. Conclusion

The syntax and semantics described here are the same as that described in [3] with the following exception:

> The `Members` function may take either a set or a sequence object as an argument and returns the collection of objects in that set or sequence. Originally, `Members` took only a set as argument.
>
> The `Size` function may take an object collection as an argument as well as sets and sequences. Originally, `Size` took only a set or a sequence as argument.
>
> The **In** operator may take a collection or sequence as its right operand as well as a set. Originally, **In** took only a set as right operand.
>
> The meaning of the Type function within a process assertion was not originally made clear. It does not make sense to give it the same meaning as within a structure. Therefore, the semantics of Type within a process assertion has been changed. Within a process assertion, Type returns the type of the objects in its argument. Type may be used with the binary operator Same to assert that an object is the same as a given type. We feel that this "fix" is inelegant, and recommend that a more permanent change be made (see chapter 13).

Appendix B

IDL Specifications

This appendix contains the IDL specifications for the three processes involved in assertion checking: semantic analysis, code generation, and interpretation. Each specification was translated with the IDL Translator [5] to create code and macros used in the implementation of the assertion checker.

1. Semantic Analysis

```
-- include the intermediate symbol table produced by the IDL
-- translator
#include "/usr/softlab/src/idlc/spec/IDLIntSymbolTable.idl"

-- IDL Symbol Table produced by IDL translator after semantic
--    analysis of assertions
--

Structure IDLSymbolTable Root scope
      From IDLIntSymbolTable Is

      Without           definition => formals,
                  IDLdefinition => body,
                  privateDefinition => returnType;


-- Add a set to store definitions

      structure_process => defStore : Set Of definition;


-- Add a new definition structure


      definition =>   overload : Set Of Instance,
                  deftype : typeTree;

      Instance ::= idlInstance | privateInstance;

      Instance => formals : Seq Of formal;

      idlInstance => body : expression;

      privateInstance => returnType : typeTree;


-- Add type attribute to all expressions

      expression => exp_type : typeTree;

-- Add new types of expressions

      expression ::= control | defnRef | formArg ;

-- control variable for a quantified expression

      control => owner : quantifier,
               name  : String;

-- name which references a definition
-- if the instance can be specified, do so

      defnRef => name : String,
               reference : DefOrInst,
                 arguments : Seq Of expression;

        DefOrInst ::= definition | Instance;

-- a formal argument in a definition

        formArg => name : String,
                 actual : formal;

-- Add collection types to typeTree

        typeTree ::= collection | notype;

        collection ::= singleton | arbitrary;

        collection => object_type : typeTree;

        singleton =>;  arbitrary =>;

        notype =>:


-- Add structure for supplied functions, which have been
--      separated out from other applications

        expression ::= SuppliedFunc;

        SuppliedFunc ::= members | head | type | size | tail;
              members=>; head=>; type=>; size=>; tail=>;

        SuppliedFunc => argument : expression;


--            ASSERTIONS

-- The type of an assertion body expression must be boolean
        Assert ForAll A In assertion Do
              Type(A.body.exp_type) Same bool Od;

-- The sets of all quantifiers must have type object collection
        Assert ForAll Q In quantifier Do
              Type(Q.set.exp_type) Sub collection Od;

-- The body of all quantifiers must be of type boolean
        Assert ForAll Q In quantifier Do
              Type(Q.body.exp_type) Same bool Od;


-- All applications must have the same name as some definition
        Assert ForAll App In application Do
              Exists D In definition Do
                    D.name.name = App.name.name Od
              Od;

-- Supplied functions must have certain kinds of arguments

        Assert ForAll func In SuppliedFunc Do
```

```
          If Type(func) Same members Then
                Type(func.argument.exp_type) Same set Or
                Type(func.argument.exp_type) Same seq
          OrIf Type(func) Same head Then
                effective_type(func.argument.exp_type) Same seq
          OrIf Type(func) Same size Then
                Type(func.argument.exp_type) Sub collection Or
                effective_type(func.argument.exp_type) Same str Or
                effective_type(func.argument.exp_type) Same seq Or
                effective_type(func.argument.exp_type) Same set
          OrIf Type(func) Same tail Then
                effective_type(func.argument.exp_type) Same seq
          Else True  Fi
          Od;


-- The test expression of all conditionals must have type boolean
      Assert ForAll C In conditional Do
          Type(C.test.exp_type) Same bool Od;

      Assert ForAll EP In expressionPair Do
          Type(EP.test.exp_type) Same bool Od;


-- The expressions following 'then' and 'else' in all conditionals
--    must have the same type
      Assert ForAll C In conditional Do
          Type(C.then.exp_type) Same Type(C.otherwise.exp_type) Od;

      Define OrIfType(OF:Seq Of expressionPair) =
          If Size(OF) = 0 Then
                Empty
          Else Type(Head(OF).then.exp_type) Union
               OrIfType(Tail(OF)) Fi;

      Assert ForAll C In conditional Do
          If Size(C.orif) > 0 Then
                Type(C.then.exp_type) Same OrIfType(C.orif)
          Else True Fi Od;


-- The type of the entire conditional should be the same as that
--    of the expression following the 'then'
      Assert ForAll C In conditional Do
          Type(C.exp_type) Same Type(C.then.exp_type) Od;


-- The following assertions involve correct types in binary and
--    unary expression operands

      Define Numeric       = int Union rat;
      Define NumOrString   = Numeric Union str;
      Define NumUnaryOps   = UnaryPlus Union UnaryMinus;
      Define CollectionOps = union Union intersect Union same Union
                       subset Union propSubset;
      Define ValueOps       = ArithmeticOps Union BoolOps Union
                      RelationalOps Union NumUnaryOps;
```

```
    Define BoolOps       = and Union or;
    Define ArithmeticOps = plus Union minus Union times Union
                           divide;
    Define RelationalOps = less Union lessEq Union greater Union
                           grtrEq Union equal Union notEqual;
.

-- Types operands of binary expression must have
-- certain types
    Assert ForAll B In binary Do
           If Type(B.op) Sub BoolOps Then
               effective_type(B.left.exp_type) Same bool And
               effective_type(B.right.exp_type) Same bool
           OrIf Type(B.op) Sub RelationalOps Then
               effective_type(B.left.exp_type) Sub NumOrString And
               effective_type(B.right.exp_type) Sub NumOrString
           OrIf Type(B.op) Sub CollectionOps Then
               Type(B.left.exp_type) Sub collection And
               Type(B.right.exp_type) Sub collection
           OrIf Type(B.op) Same inSet Then
               ((effective_type(B.right.exp_type) Same set Or
                effective_type(B.right.exp_type) Same seq) And
               effective_type(B.left.exp_type) Same
               component_type(B.right.exp_type))
               Or
               Type(B.right.exp_type) Same arbitrary
           OrIf Type(B.op) Sub ArithmeticOps Then
               effective_type(B.left.exp_type) Sub Numeric And
               effective_type(B.right.exp_type) Sub Numeric
           Else True
           Fi Od;

-- Types arguments of unary expressions must have
    Assert ForAll U In unary Do
           If Type(U.op) Sub NumUnaryOps Then
                 effective_type(U.body.exp_type) Sub Numeric
           Else effective_type(U.body.exp_type) Same bool Fi Od;


-- effective type is the type of the actual object(s).
-- references are tracked down to get the type of the referenced object

    Define effective_type(t:basic)          = Type(t);
    Define effective_type(t:SetOrSeq)    = Type(t);
    Define effective_type(t:user)        = Type(t);
    Define effective_type(t:arbitrary)   = Type(t);
    Define effective_type(t:singleton)   = Type(t.object_type);
    Define effective_type(t:notype)      = Type(t);


-- component type is the type of the component of a set or sequence
-- empty if not a set or sequence

    Define component_type(t:SetOrSeq)    = Type(t.component);
    Define component_type(t:basic)          = Empty;
    Define component_type(t:user)        = Empty;
    Define component_type(t:arbitrary)   = Empty;
```

```
      Define component_type(t:singleton)   = component_type(t.object_type);
      Define component_type(t:notype)      = Empty;

-- A collection may be an operand of a binary value operator if it
-- is assured of containing a single object

      Assert ForAll B In binary Do
          If Type(B.op) Sub ValueOps Then
            is_single_object(B.left) And
            is_single_object(B.right)
          Else True Fi
            Od;

-- A collection may be an operand of a unary opeator if it
-- is assured of containing a single object

      Assert ForAll U In unary Do
          is_single_object(U.body) Od;



-- definition of what it means to be a single object or a
-- collection assured of containing a single object

      Define is_single_object(x:expression) =
          Not (Type(x.exp_type) Same arbitrary);



-- Two sequences of formal arguments are distinct if their heads are
-- distinct and their tails are distinct

      Define Distinct(a:Seq Of formal,b:Seq Of formal) =
            If Size(a) ~= Size(b) Then
                  True
            Else If Size(a) >= 1 Then
                        Distinct(Head(a),Head(b)) Or Distinct(Tail(a),Tail(b))
                 Else False
                 Fi
            Fi;


-- Two formal arguments are distinct if the type of neither is
-- included in the other

      Define Distinct(a:formal,b:formal) =
            Not ( (Type(a.type) Sub Type(b.type)) Or
               (Type(b.type) Sub Type(a.type)) );


-- Different instances of the same definition must be distinguishable
-- by the number or types of their formal arguments

      Assert ForAll D In definition Do
            ForAll I1 In Members(D.overload) Do
                  ForAll I2 In Members(D.overload) Do
                        If I2 ~= I1 Then
```

```
                                       Distinct(I1.formals,I2.formals)
                                   Else True Fi
                                   Od
                             Od
                       Od;

            End  -- IDLSymbolTable


            Structure SemAssert_inv
                  From IDLSymbolTable IDLIntSymbolTable Is


                  structure_process => quantStack : Seq Of quantifier,
                                    defseq : Seq Of definition,
                                    invariants : Set Of structure;

                  -- Add unknown type to typeTree
                  -- (used to prevent cascading error messages

                  typeTree ::= unknown;

                  unknown =>;

                  -- structures to handle cyclic graph search
                  -- and reverse topological sort of noncyclic definitions

                  definition => calls : Set Of definition,
                              revcalls : Set Of definition,
                              cycle : Boolean,
                              visit : Boolean,
                              index : Integer;

            End  -- SemAssert_inv



            Process SemanticAssert Inv SemAssert_inv Is

                  Target C;
                  Pre   IntSymbolTable : IDLIntSymbolTable;
                  Post  SymbolTable    : IDLSymbolTable;

            End -- SemanticAssert
```

2. Code Generation

```
-- include the symbol table, the intermediate symbol table
#include "../seman/IDLSymbolTable.idl"
#include "/usr/softlab/src/idlc/specs/IDLIntSymbolTable.idl"


Process GeneratePostfix Is

      Target C;
      Pre   SymbolTable : IDLSymbolTable;
      Post  PostfixCode : Postfix;

      Restrict * To Create, Destroy, foreachinSEQ, foreachinSET,
            inSET, retrievelastSEQ, removelastSEQ, appendrearSEQ;

End  -- GeneratePostfix



Structure Postfix Root scope
      From IDLSymbolTable Is

-- Add a nesting level attribute to quantifiers
--  used by code generator

      quantifier => nest_level : Integer;

-- Add a postfix code version of expression body to assertions

      assertion => postfixBody : Seq Of Integer;

-- When available, make code an array
-- For assertion.postfixBody Use array;

-- Add a postfix code version of expression body to definition
--    instances

      idlInstance => postfixDefn : Seq Of Integer;


-- Postfix code entry is an integer code for an instruction

-- When available, make code an array
-- For idlInstance.postfixDefn Use array;

-- Add an integer attribute to each expression which points to
-- position in runtime code array where expression's code begins
-- ( used in printing out error messages on non-satisfied assertions

      expression => valuepos : Integer;

-- Each structure and process has arrays holding strings, integers,
-- and rationals refered to by instructions

      structure_process => string_refs   : Seq Of String,
                        integer_refs  : Seq Of Integer,
                        rational_refs : Seq Of Rational,
                        type_refs     : Seq Of typeTree,
                        define_refs   : Seq Of definition;

-- When available, make reference lists arrays
--For structure_process.string_refs Use array;
--For structure_process.integer_refs Use array;
--For structure_process.rational_refs Use array;
--For structure_process.type_refs Use array;
--For structure_process.define_refs Use array;

End  -- Postfix
```

3. Interpretation

```
-- Include symbol table, intermediate symbol table, code
-- generation structures and idldata (structure instance)
#include "../seman/IDLSymbolTable.idl"
#include "/usr/softlab/src/idlc/specs/IDLIntSymbolTable.idl"
#include "../codegen/gencode.idl"
#include "/usr/softlab/src/specs/idldata.idl"

Structure check_inv Root inv_root
      From Postfix IDLdata Is

      inv_root => pfcode : scope,
                collection_store : Seq Of collection,
                runstackentries : Seq Of runstackEntry;
--              struc_store : PortStruc;

      runstackEntry ::= collection | IDLVALUE | TVALUE | FVALUE ;

      collection => objects : Seq Of IDLVALUE;

      TVALUE =>; FVALUE =>;

End  -- check_inv

Process Check Inv check_inv Is

      Target C;
      Pre   Code      : Postfix;
      Pre   data_in    : IDLdata;
--    Post  an error log printed to standard out or designated file

      Restrict * To Create, Destroy, addSET, foreachinSET,
              ithinSEQ, inSET, emptySET;

      Restrict runstackEntry To Create, Destroy, foreachinSEQ,
            ithinSEQ, inSEQ, retrievefirstSEQ, appendfrontSEQ,
            removefirstSEQ;
      Restrict collection To Create, Destroy, foreachinSEQ,
            ithinSEQ, inSEQ, appendrearSEQ, removelastSEQ ,
            retrievelastSEQ, emptySEQ;
      Restrict IDLVALUE To Create, Destroy, foreachinSEQ,
            ithinSEQ, inSEQ, appendrearSEQ, removelastSEQ,
            foreachinSET, inSET, retrievefirstSEQ,
            emptySEQ;
      Restrict PortStruc To Create, Destroy, foreachinSEQ,
            appendrearSEQ;
      Restrict attrDesc To Create, Destroy, foreachinSEQ,
            foreachinSET, ithinSEQ, inSEQ, inSET, appendrearSEQ,
            removelastSEQ ;
      Restrict NT To Create, Destroy, foreachinSEQ;
      Restrict SYMBOL To Create, Destroy, foreachinSEQ;
      Restrict assertionStatement To Create, Destroy, foreachinSEQ;
      Restrict formal To Create, Destroy, ithinSEQ, foreachinSEQ;
```

End -- Check

Appendix C

Interpreter Virtual Machine Instructions

This appendix describes individual interpreter machine instructions. The instructions are grouped according to the nature of their operation and the type of their operands. On the left is the name of the instruction. The number preceeding the instruction is the integer code for that instruction. On the right is a verbal description of the instruction. Below the instruction name is a notational description of the runtime stack before and after execution of the instruction. Only the relevant portion of the stack (the top few elements) is shown. Some instructions have attributes as well as stack operands. Attributes, with type specified after a colon, are in parentheses to the right of the instruction. Attributes are represented in the code array as two byte integers, which index an array containing the attribute itself. In the remainder of this appendix, whenever *stack* is mentioned, only the relevant portion at the top of the evaluation stack is meant.

The stack before execution is represented on the left of the description. This is followed by a colon (:), followed by a representation of the stack after execution of the instruction. Each representation of the stack is enclosed within angle brackets (<>). Within brackets, the stack grows towards the top from left to right. Individual operands are separated by commas (,) and vertical bars (|) indicate exclusive alternatives (one or the other value, but not both). The operand closest to the right bracket (>) is the top-of-stack (TOS). The operand to the left of TOS is TOS-1. Brackets that do not enclose any operands represent an empty evaluation stack.

The following type abbreviations are used:

int	integer
rat	rational
str	string
bool	boolean
set	set
seq	sequence
node	node
coll	collection
num	int | rat
value	int | rat | str | bool | set | seq | node
def	definition
instance	instance of a defintion
type	num | bool | str | node | set of <type> | seq of <type>

The format of this description is taken from the desciption of the UCSD Pascal P-code machine [1].

Literals

0 integerzero	Push the integer 0
1 integerone	Push the integer 1
2 integer(*i*:int) <>:<int>	Push the integer *i*
3 rational(*r*:rat) <>:<rat>	Push the rational *r*
4 string(*s*:str) <>:<str>	Push the string *s*
5 true <>:<bool>	Push TRUE
6 false <>:<bool>	Push FALSE
7 empty <>:<coll>	Push the empty collection
8 root <>:<node>	Push the collection containing only the root node object of the structure in which root appears.
9 typeExpression(*t*:type) <>:<coll>	Push the collection containing all objects with the type specified by *t*. The objects are taken from the structure in which the typeExpression is found.

Collection Infix Operators

10 root(*portname*:str) <>:<node>	Push the collection containing only the root node object of the structure associated with the port specified by *portname*.
11 portExpression(*portname*:str,*t*:type) <>:<coll>	Push the collection containing all objects with the type specified by *t*. The objects are taken from the structure associated with the port specified by *portname*.

Collection Binary Operators

12 union <coll,coll>:<coll>	Collection union. Push the union of collections TOS and TOS-1.
13 intersect <coll,coll>:<coll>	Collection intersection. Push the intersection of collections TOS and TOS-1.

14 subset <coll,coll>:<bool>	Collection subset. Push the boolean result of TOS-1 is subset of TOS.
15 propsubset <coll,coll>:<bool>	Collection proper subset. Push the boolean result of TOS-1 is proper subset of TOS.

Arithmetic Binary Operators

16 plus <num,num>:<num>	Addition. Add TOS into TOS-1 and push the result. If either of the operands is rational, the result will be rational. Otherwise, the result will be integer.
17 minus <num,num>:<num>	Substraction. Subtract TOS from TOS-1 and push the result. If either of the operands is rational, the result will be rational. Otherwise, the result will be integer.
18 times <num,num>:<num>	Multiplication. Multiply TOS into TOS-1 and push the result. If either of the operands is rational, the result will be rational. Otherwise, the result will be integer.
19 divide <num,num>:<num>	Division. Divide TOS into TOS-1 and push the result. If either of the operands is rational, the result will be rational. Otherwise, the result will be integer.

Boolean Binary Operators

20 and <bool,bool>:<bool>	Boolean AND. Push the boolean result of TOS-1 AND TOS.
21 or <bool,bool>:<bool>	Boolean OR. Push the boolean result of TOS-1 OR TOS.

Relational Operators on Strings

22 str_less <str,str>:<bool>	Less-than relation. If TOS-1 is lexicographically less than TOS, as defined by the standard ASCII ordering, push TRUE. Otherwise push FALSE.
23 str_lessEq <str,str>:<bool>	Less-than-or-equal relation. If TOS-1 is lexicographically less than or equal to TOS, as defined by the standard ASCII ordering, push TRUE. Otherwise, push FALSE.
24 str_greater <str,str>:<bool>	Greater-than relation. If TOS-1 is lexicographically greater than TOS, as defined by the standard ASCII ordering, push TRUE. Otherwise push FALSE.

25 str_grtrEq <str,str>:<bool>	Greater-than-or-equal relation. If TOS-1 is lexicographically greater than or equal to TOS, as defined by the standard ASCII ordering, push TRUE. Otherwise push FALSE.
26 str_equal <str,str>:<bool>	Equivalence relation. If TOS-1 is lexicographicaly equivalent to TOS, as defined by the standard ASCII ordering, push TRUE. Otherwise push FALSE.
27 str_notEqual <str,str>:<bool>	Non-equivalence relation. If TOS-1 is not lexicographically equivalent to TOS, as defined by the standard ASCII ordering, push TRUE. Otherwise push FALSE.

Relational Operators on Integers and Rationals

28 num_lessEq <num,num>:<bool>	Less-than-or-equal relation. Push the boolean result of TOS-1 ≤ TOS.
29 num_less <num,num>:<bool>	Less-Than relation. Push the boolean result of TOS-1 < TOS.
30 num_greater <num,num>:<bool>	Greater-than relation. Push the boolean result of TOS-1 > TOS.
31 num_grtrEq <num,num>:<bool>	Greater-than-or-equal relation. Push the boolean result of TOS-1 ≥ TOS.
32 num_equal <num,num>:<bool>	Equivalence relation. Push the boolean result of TOS-1 = TOS.
33 num_notEqual <num,num>:<bool>	Non-equivalence relation. Push the boolean result of TOS-1 ≠ TOS.

Relational Operators on Booleans

34 bool_equal <bool,bool>:<bool>	Equivalence relation. If TOS and TOS-1 are both TRUE or both operands are FALSE, push TRUE. Otherwise push FALSE.
35 bool_notEqual <bool,bool>:<bool>	Non-equivalence relation. If TOS and TOS-1 are both TRUE or both operands are FALSE, push FALSE. Otherwise push TRUE.

Relational Operators on Sets

36 set_equal <set,set>:<bool>	Equivalence relation. If TOS and TOS-1 contain exactly the same objects, push TRUE. Otherwise push FALSE.

37 set_notEqual <set,set>:<bool>	Non-equivalence relation. If TOS and TOS-1 do not contain exactly the same objects, push TRUE. Otherwise push FALSE.

Relational Operators on Sequences

38 seq_equal <seq,seq>:<bool>	Equivalence relation. If TOS and TOS-1 contain exactly the same objects in the same order, push TRUE. Otherwise push FALSE.
39 seq_notEqual <seq,seq>:<bool>	Non-equivalence relation. If TOS and TOS-1 do not contain exactly the same objects in the same order, push TRUE. Otherwise push FALSE.

Relational Operators on Node Objects

40 node_equal <node,node>:<bool>	Equivalence relation. If TOS and TOS-1 are the same object, push TRUE. Otherwise push FALSE.
41 node_notEqual <node,node>:<bool>	Non-equivalence relation. If TOS and TOS-1 are not the same object, push TRUE. Otherwise push FALSE.

Miscellaneous Relational Operators

42 same <coll,coll>:<bool>	Equivalence relation for collections. If TOS and TOS-1 contain exactly the same objects, push TRUE. Otherwise push FALSE.
43 inSet <num | str | bool | node,set>:<bool>	Set inclusion. If TOS-1 is a member of TOS, push TRUE. Otherwise push FALSE.
44 inSeq <num | str | bool | node,seq>:<bool>	Sequence inclusion. If TOS-1 is a member of TOS, push TRUE. Otherwise push FALSE.
44 inCollection <num | str | bool | node,coll>:<bool>	Collection inclusion. If TOS-1 is a member of TOS, push TRUE. Otherwise push FALSE.

Unary Operators

46 unaryMinus <num>:<num>	Unary minus. Multiply TOS by -1 and push the result. The type of TOS will remain the same.
47 not <bool>:<bool>	Boolean negation. If TOS is TRUE, push FALSE. If TOS is FALSE, push TRUE.

Jumps

48 jump(*loc*:int) <>:<>	Jump to the location in the code array indexed by *loc*.

49 jtrue(*loc*:int) <bool>:<>	Jump to the location in the code array indexed by *loc* if TOS is TRUE.
50 jfalse(*loc*:int) <bool>:<>	Jump to the location in the code array indexed by *loc* if TOS is FALSE.

System Function Calls

51 members <set | seq>:<coll>	Push the collection containing the objects contained in TOS.
52 head <seq>:<num | str | bool | node>	Push the first object in TOS.
53 tail <seq>:<seq>	Push the sequence obtained by deleting the first in object in TOS.
54 str_size <str>:<int>	Push the number of characters in TOS.
55 setorseqorcoll_size <set | seq>:<int>	Push the number of objects contained in TOS.
56 type <coll>:<coll>	For each object in TOS, create a collection containing all objects of the same type as the object in TOS. Push the union of these collections.

Quantifier Operators

57 forall <coll>:<>	Remove the first object of TOS and place it in the control stack at the current offset. Increment the offset. Insert the modified TOS onto the end of the quantifier collections list.
58 exists <coll>:<>	Remove the first object of TOS and place it in the control stack at the current offset. Increment the offset. Insert TOS onto the end of the quantifier collections list.
59 endForAll(*return*:int) <bool>:<bool> | <>	If TOS is FALSE, remove the last entry of the quantifier collections list and push FALSE. If TOS is TRUE and the last entry of the quantifier collections list is empty, remove the last entry of the quantifier collections list and push TRUE. If TOS is TRUE and the last entry of the quantifier collections list is non-empty, remove the first object of the last entry of the quantifier collections list, place it in the control stack at the current offset-1, and jump to the position in the code indexed by *return*.

60 endExists(*return*:int) <bool>:<bool> | <>	If TOS is TRUE, remove the last entry of the quantifier collections list and push TRUE. If TOS is FALSE and the last entry of the quantifier collections list is empty, remove the last entry of the quantifier collections list and push FALSE. If TOS is FALSE and the last entry of the quantifier collections list is non-empty, remove the first object of the last entry of the quantifier collections list, place it in the control stack at the current offset-1, and jump to the position in the code indexed by *return*.
61 control(*level*:int) <>:<value>	Push the object located in the control stack indexed by current offset + *level*.
62 formArg(*pos*:int) <>:<value>	Push the object located in the definition's argument list at position *pos*.

Miscellaneous Operators

63 dot(*attr*:str) <coll>:<coll>	Push the collection containing all objects that are associated with the attribute (specified by the name attr) of all objects in TOS. For each object in TOS, there will be exactly one object in the result collection.
64 application(*ref*:instance | def,*i*:int) <{value}^{i}>:<value | coll>	If *ref* is of type def, then determine which instance of the referenced definition applies to this application. Instances are determined by matching number and types of actual arguments with number and types of formal arguments. Evaluate the user-defined definition instance specified by *ref*, applied to the correct number of arguments (specified by *i*) at the top of the stack. Push the result returned by the instance evaluation.
65 return <>:<>	Signals end of a definition instance evaluation. Return TOS.
66 endAssertion <bool>:<>	If TOS is TRUE, the assertion is true. If TOS is FALSE, the assertion is false. Report the result.

APPENDIX D

UNIX Manual Pages and Shell Scripts

This appendix contains the Unix manual pages and the Csh shell scripts for the *assertcodegen* program, which generates code for the assertion checker, and the *idlcheck* program, which interprets assertions given generated code and particular instances.

NAME

assertcodegen - the IDL assertion code generator

SYNOPSIS

assertcodegen [*option*]

DESCRIPTION

assertcodegen generates the code which the assertion checker(*idlcheck*) will execute. Input is the symbol table generated by the IDL translator with the -s option (idlc -s *symboltable*). The symbol table is assumed to be from stdin unless the -i option is used.

Output is a file containing appropriate postfix code which the checker can interpret, as well as the original symbol table. The structure of this output will conform to the IDL specifications in assertcontrol(5-IDL). The output goes to stdout unless the -o option is used.

OPTIONS

-i *filename*

The symbol table is to be found in *filename*

-o *filename*

The output is to go to *filename*

SEE ALSO

Kickenson, J. *A Tutorial to the IDL Assertion Language*

idlc(1-IDL), idlcheck(1-IDL), assertcontrol(5-IDL)

FILES

/usr/softlab/bin/assertcodegen

-the IDL assertion code generator

/usr/softlab/bin/idlcheck

-the IDL assertion checker

/usr/softlab/src/assertcodegen/specs/assertcontrol

-the IDL specifications for the assertcodegen process

AUTHORS

Jerry Kickenson, Richard Snodgrass

University of North Carolina at Chapel Hill

DIAGNOSTICS

The diagnostics produced by *assertcodegen* are intended to be self-explanatory.

NAME

idlcheck – the interface description language (IDL) assertion checker

SYNOPSIS

idlcheck [*option*] ... [*portname:structure_instance*]... *assertioncontrol*

DESCRIPTION

assertioncontrol is a file containing the assertion control code produced by the assertion code generator (*assertcodegen*). If *assertioncontrol* is '-', the assertion control code may be found in standard input.

The input and/or output structure instances of a particular run of a process must be indicated by specifying the name of a file containing the external representation of an instance. The user must also indicate which port of his process each structure instance is associated with. This is done by separating the portname from the structure instance by a colon. There may be any number of input or output instances, except that there must be at least one instance (input or output). If '-' appears after the colon, this indicates that the structure instance associated with the port is found in standard input. Multiple instances may be designated with '-'. Each instance will be read in order from standard input.

The input and output must be in the ASCII external representation language. Files produced by IDL processes will be in this form.

The checker will produce an error log indicating any false assertions, with diagnostics to aid in pinpointing the errors. The error log is printed to stdout unless the -e option is used.

OPTIONS

-c *filename*

Read the generated code from the file *filename*

-e *filename*

Write the error log to *filename* rather than to stdout.

-w

Suppress error messages. Write out only satisfied/not satisfied.

-P *privatedefs*

The bodies of private definitions are found in *privatedefs*.

EXAMPLES

Assume specs contains the IDL specifications for the process:

```
Process example Is
   Pre:  in1  : structure_in;
   Pre:  in2  : structure_in;
   Post: out1 : structure_out;
   Post: out2 : structure_out;
End
```

Assume that the input structure instances for the specific run are in I1 and I2 and that the output structure instances for the specific run are in O1 and O2.

1. (using files)

```
idlc specs -s symboltable
assertcodegen symboltable -o assertioncontrol
idlcheck in1:I1 in2:I2 out1:O1 out2:O2 assertioncontrol
```

2. (using pipes)

```
idlc specs -s - | assertcodegen | idlcheck in1:I1 in2:I2 out1:O1 out2:O2 -
```

SEE ALSO

Kickenson, J. *A Tutorial to the IDL Assertion Language*
idlc(1-IDL), assertcodegen(1-IDL), assertcontrol(5-IDL)

BUGS

Private definitions are not supported.

AUTHORS

Jerry Kickenson
University of North Carolina at Chapel Hill

DIAGNOSTICS

The diagnostics produced by *idlcheck* are intended to be self-explanatory.

```
#!/bin/csh -f
# Shell script for assertcodegen program

set usage = "usage: assertcodegen [-i symboltable] [-o generated_code]"
set oflag
set ofile
set iflag
set ifile

set gen = /usr/softlab/src/assertcodegen/gen

while  ($#argv > 0)

    switch ($1)
        case -i:
            set iflag = "-i"
            shift
            if ($#argv > 0) then
                set ifile = $1
            else
                echo "$usage"
                exit 1
            endif
            shift
            breaksw

        case -o:
            set oflag = "-o"
            shift
            if ($#argv > 0) then
                set ofile = $1
                /bin/rm -f $ofile
            else
                echo "$usage"
                exit 1
            endif
            shift
            breaksw

        default:
            echo "$usage"
            exit 1

    endsw
end

if (("$iflag" == "-i") && ("$oflag" == "-o")) then
        $gen < $ifile > $ofile
else if ("$iflag" == "-i") then
        $gen < $ifile
     else if ("$oflag" == "-o") then
              $gen > $ofile
          else
              $gen
endif

exit $status
```

```
#!/bin/csh -f
#Shell script for idlcheck program

set echo
set usage = "usage: idlcheck [option]... [port:structure]... -c code"
set eflag
set efile
set cflag
set cfile
set strucs = ""
set temp

set chk  = /usr/softlab/src/idlcheck/check

while  ($#argv > 0)

    switch ($1)
       case -c:
           set cflag = "-c"
           shift
           if ($#argv > 0) then
               set cfile = $1
           else
               echo "$usage"
               exit 1
           endif
           shift
           breaksw

       case -e:
           set eflag = "-e"
           shift
           if ($#argv > 0) then
               set efile = $1
               /bin/rm -f $efile
           else
               echo "$usage"
               exit 1
           endif
           shift
           breaksw

       case -w:
           set strucs = "$strucs $1"
           shift
           breaksw

       default:
           if($1 =~ *:*) then
               set temp = `echo $1 | sed 's/:/ /g'`
               set strucs = "$strucs $temp"
           else
              echo "$usage"
              exit 1
           endif
           shift
    endsw
```

```
end

if ("$cflag" == "-c" && "$eflag" == "-e") then
        $chk $strucs < $cfile >& $efile
else if ("$cflag" == "-c") then
        $chk $strucs < $cfile
     else if ("$eflag" == "-e") then
              $chk $strucs >& $efile
          else
              $chk $strucs
endif

exit $status
```

APPENDIX E

Example Checker Output

This appendix contains a full example IDL specification with assertions and an actual output of the assertion checker on specific instances of the input and output structures. In order to illustrate the assertion failure log, the instances were known to have errors.

```
-- Example IDL specification

Structure customers Root customer_list Is

      -- There is at least one customer
A1    Assert Size(Root.list) ~= 0;

      customer_list  =>   list : Seq Of Customer;

      Customer        ::=   commercial_customer
                        | Government_customer;
      Customer        =>   name              : String,
                       address          : String,
                       active           : Boolean,
                       customer_number  : Integer,
                       balance          : Rational;

A2    Assert ForAll C In Customer Do C.customer_number = 1 Od;
A3    Assert Exists C In Customer Do C.active = True Od;

      -- No duplicate customer numbers
A4    Assert ForAll c1 In Customer Do
            ForAll c2 In Customer Do
                  If c1 ~= c2 Then
                      c1.customer_number ~= c2.customer_number
                  Else True Fi
            Od Od;

A5    Assert Exists C In Customer Do C.name = "IBM" Od;

      commercial_customer =>   industry_code    : Integer;

      Government_customer ::=    state_customer
                            | federal_customer;

      state_customer       =>   state_code : Integer;

A6    Assert ForAll sc In state_customer Do
            sc.state_code >= 1 And sc.state_code <= 50 Od;

      federal_customer     =>   agency_code : Integer;

End

Structure transactions Root transaction_list Is
```

```
      transaction_list    =>   list : Seq Of Transaction;

      Transaction           ::=  credit | debit;
      Transaction         =>   customer_number : Integer,
                         date    .        : Integer,
                                amount           : Rational,
                         tax_status : Set Of Tax_code;

T1    Assert ForAll t In Transaction Do t.customer_number >= 1 Od;

      credit              =>   ;
      debit               =>   ;

      Tax_code            ::=    local_sales_tax
                         | state_sales_tax
                         | federal_sales_tax;

      local_sales_tax     =>   ;
      state_sales_tax     =>   ;
      federal_sales_tax   =>   ;


End

Structure bills Root bill_list Is

      bill_list       =>   list : Seq Of bill;

      bill            =>   billee : Customer,
                       amount : Rational;

B1    Assert ForAll b In bill Do b.amount > 0 Od;

      Customer        ::=    commercial_customer
                       | Government_customer;
      Customer        =>   name               : String,
                       address          : String,
                       customer_number : Integer;

      commercial_customer =>   industry_code    : Integer;

      Government_customer ::=    state_customer
                            | federal_customer;

      state_customer      =>   state_code : Integer;

      federal_customer    =>   agency_code : Integer;

End

Process billing Is
      Target C;
      Pre   customers_in    : customers;
      Pre   transactions_in : transactions;
      Post  customers_out   : customers;
      Post  bills_out       : bills;
      -- Customers output match customers input
P1    Assert ForAll c_in In customers_in:Customer Do
          ForAll c_out In customers_out:Customer Do
            If c_in.customer_number = c_out.customer_number Then
                c_in.name ~= c_out.name And
                c_in.address = c_out.address And
                c_in.active = c_out.active
            Else True Fi
         Od Od;

P2    Assert ForAll cc_in In customers_in:commercial_customer Do
          ForAll cc_out In customers_out:commercial_customer Do
            If cc_in.customer_number = cc_out.customer_number Then
                cc_in.industry_code = cc_out.industry_code
            Else True Fi
          Od Od;

      -- Each transaction refers to an actual customer
P3    Assert ForAll t In transactions_in:Transaction Do
          Exists c In customers_in:Customer Do
            t.customer_number = c.customer_number
          Od Od;


      Define Total_Credit(c:Customer, TList:Seq Of Transaction) =
          If Size(TList) = 0 Then 0
          OrIf Head(TList).customer_number = c.customer_number
               And Type(Head(TList)) Same transactions_in:credit
            Then Head(TList).amount + Total_Credit(c,Tail(TList))
          Else Total_Credit(c,Tail(TList))
          Fi;


      -- All customers are properly credited
P4    Assert ForAll c_out In customers_out:Customer Do
          ForAll c_in In customers_in:Customer Do
            If c_in.customer_number = c_out.customer_number Then
                c_out.balance = c_in.balance +
                  Total_Credit(c_in, transactions_in:Root.list)
            Else True Fi
          Od Od;

      Define Num_debits(TList: Seq Of Transaction) =
            If Size(TList) = 0 Then 0
            OrIf Type(Head(TList)) Same transactions_in:debit Then
                  1 + Num_debits(Tail(TList))
            Else Num_debits(Tail(TList)) Fi;

P5    Assert Size(bills_out:Root.list) =
            Num_debits(transactions_in:Root.list);

P6    Assert Size(bills_out:Root.list) = Size(transactions_in:debit);

      Define Debits(TList: Seq Of Transaction) =
            If Size(TList) = 0 Then Empty
            OrIf Type(Head(TList)) Same transactions_in:debit Then
                  Head(TList) Union Debits(Tail(TList))
            Else Debits(Tail(TList)) Fi;

      -- A bill is generated for every debit transaction
P7    Assert ForAll deb In Debits(transactions_in:Root.list) Do
            Exists b In Members(bills_out:Root.list) Do
                  b.billee.customer_number = deb.customer_number And
                  b.amount = deb.amount
            Od Od;

P8    Assert Size(bills_out:Root.list)
            = Size(Debits(transactions_in:Root.list));


End  -- billing
```

Output of the checker:

```
Assertion A1 in customers_in is true.
Assertion A1 in customers_out is true.
Assertion A2 in customers_in is false.
> FORALL is not satisfied when C is L3
  > Customer is { L4 L3 L2 }
  & Equality not satisfied.
    > .customer_number is { 2 }
      > C is { L3 }
Assertion A2 in customers_out is false.
> FORALL is not satisfied when C is L12
  > Customer is { L13 L12 L11 }
  & Equality not satisfied.
    > .customer_number is { 2 }
      > C is { L12 }
Assertion A3 in customers_in is true.
Assertion A3 in customers_out is true.
Assertion A4 in customers_in is true.
Assertion A4 in customers_out is true.
Assertion A5 in customers_in is true.
Assertion A5 in customers_out is true.
Assertion A6 in customers_in is true.
Assertion A6 in customers_out is true.
Assertion T1 in transactions_in is true.
Assertion B1 in bills_out is true.
Assertion P1 is false.
> FORALL is not satisfied when c_in is L2
  > Customer is { L4 L3 L2 }
  & FORALL is not satisfied when c_out is L11
    > Customer is { L13 L12 L11 }
    & Conditional is not satisfied.
      > Equality is satisfied.
        > .customer_number is { 1 }
          > c_in is { L2 }
        & .customer_number is { 1 }
          > c_out is { L11 }
      & And not satisfied.
        > And not satisfied.
          > InEquality not satisfied.
            > .name is { Innovation, Inc. }
              > c_in is { L2 }
            & .name is { Innovation, Inc. }
              > c_out is { L11 }
Assertion P2 is true.
Assertion P3 is true.
Assertion P4 is true.
Assertion P5 is true.
Assertion P6 is true.
Assertion P7 is true.
Assertion P8 is true.
```

Bibliography

1. Anderson, G., Clark, R., Chapin C., Franks, B., Overgaard, M., Stringfellow, S. *UCSD p-System Internal Architecture Guide: Version 4.0.* SofTech Microsystems, Inc., San Diego, CA, (1981).

2. Goos, G. **Diana Reference Manual.** Carnegie-Mellon University, (1981).

3. J.R. Nestor, W.A Wulf, D.A. Lamb **IDL Formal Description, Part I.** SoftLab Document No. 2, University of North Carolina, (1985).

4. Lamb, David Alex **Sharing Intermediate Representations: The Interface Description Language.** Ph.D. Dissertation, Carnegie-Mellon University, (1983).

5. Shannon, K., T. Maroney, R. Snodgrass **Using IDL with C.** 6, University of North Carolina, (1985).

6. William B. Warren, Jerry S. Kickenson, Richard Snodgrass **A Tutorial Introduction to Using IDL.** SoftLab Document No. 1, University of North Carolina, (1985).

Supplementary Appendix
Program Listings

```
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*                                                                      *
*  Title:  Semantic Analysis of Assertions                   *
*  Filename:         ~kickenso/softlab/seman/symbol.c                  *
*  Author:           Jerry Kickenson <kickenso@unc>                    *
*                    Department of Computer Science                    *
*                    University of North Carolina              *
*                    Chapel Hill, NC  27514                            *
*                                                                      *
*  Copyright (C) The University of North Carolina, 1985      *
*                                                                      *
*  All rights reserved. No part of this software may be sold or  *
*  distributed in any form or by any means without the prior written     *
*  permission of the SoftLab Software Distribution Coordinator. *
*                                                                      *
*  Report problems to            softlab@unc (csnet) or                *
*                                softlab!unc@CSNET-RELAY (ARPAnet)     *
*  Direct all inquiries to the SoftLab Software Distribution   *
*        Coordinator, at the above addresses.                  *
*                                                                      *
*  Function: main driver for semantic analysis of assertions  *
*                                                                      *
*        main        x_type                                    *
*                                                                      *
******************************************************************* */


#include "SemanticAssert.h"
#include <stdio.h>
#include "macros.h"

extern FILE  *errorfile;
BOOLEAN            recursion; /* true if a body includes recursion */

main(argc,argv)
int        argc;
char       *argv[];
{
           scope                        symbolTable;        /* root of symbol table */
           SEQSYMBOL                    SSP;      /*  iterators ...       */
           SYMBOL                       SP;
           SEQassertionStatement        Sas;
           assertionStatement    as;
           SETdefinition                Sdefn;
           definition            defn;
           SETInstance                  Sin;
           Instance              inst;


           expression            x_type(); /* typing function      */
           definition            mindef(); /* noncyclic definition
                                                                 with lowest order that has
                                                                 not been typed yet    */

           SETstructure                 invs;               /* set of invariants   */

           SETdefinition                Y;
           definition            y;
           int                          p;
           int                          num_noncyclic;      /* number of noncyclic
                                                                 definitions */
           BOOLEAN                                defcall;  /* true if a definition is
                                                                   being typed                  */

           SETdefinition                Sdef,Sd;
           definition            def,d;
```

```
        String                      defname;

        int                         index;
        FILE                        *fopen();


        for(index=1;index<argc;index++)
        {
          if(streq(argv[index],"-e"))
                errorfile = fopen(argv[++index],"w");
        }

        /* read in intermediate symbol table */

        symbolTable = IntSymbolTable(stdin);

invs = NULL;

/* collect invariant structure names          */
foreachinSEQSYMBOL(symbolTable->symbols,SSP,SP)
{
  if(typeof(SP) == Kprocess)
        addSETstructure(invs,SP.Vprocess->invariant);
}

/* analyze all structures and processes which are not invariants    */
foreachinSEQSYMBOL(symbolTable->symbols,SSP,SP)
{
  if(typeof(SP) == Kstructure)
  {
  if(!inSETstructure(invs,SP.Vstructure))
  {

  /* collect definitions into defStore          */
  add_definitions(SP);

  /* collect all definitions that a definition calls */
  /* used to check that a noncyclic definition is, in fact, noncyclic          */
  /* and to prepare noncyclic definitions for topological sort */

  foreachinSETdefinition(SP.Vstructure->defStore,Sdefn,defn)
        defn.IDLclassCommon->calls = NULL;
  findcalls(SP);


  /* mark all definitions as part of a cycle or not */

  findcycles(SP);

  /* check noncyclic definitions - if in a cycle change to
        a cyclic definition and issue a message to that effect    */

  foreachinSETdefinition(SP.Vstructure->defStore,Sdef,def)
        if(typeof(def) == Knoncyclic && def.IDLclassCommon->cycle)
        {
          Warning1(800,def.IDLclassCommon->sourceposition,
                        def.IDLclassCommon->name->name);
          /* copy noncyclic attributes to a new cyclic definition */
          defname = malloc(strlen(def.IDLclassCommon->name->name)+1);
          strcpy(defname,def.IDLclassCommon->name->name);
          Sin = def.IDLclassCommon->overload;
          def.Vcyclic = Ncyclic;
          def.Vcyclic->name = NnameToken;
          def.Vcyclic->name->name = malloc(strlen(defname)+1);
          strcpy(def.Vcyclic->name->name,defname);
          def.Vcyclic->overload = Sin;
          Sdef->value = def;
          }


    /* reverse topological sort of noncyclic definitions */
    topsort(SP);

    /* make pass through noncyclic definitions                    */

    foreachinSETdefinition(SP.Vstructure->defStore,Sdefn,defn)
            defn.IDLclassCommon->visit = FALSE;

    num_noncyclic = number_noncyclic(SP.Vstructure->defStore);
    for(p = 1;p <= num_noncyclic;p++)
    {
            defn = mindef(SP.Vstructure->defStore);
            /* type noncyclic definitions - cyclic checked later */
            /* analyze bodies of definition instances            */
       foreachinSETInstance(defn.IDLclassCommon->overload,Sin,inst)
       {
                       if(typeof(inst) == KidlInstance)
                       {
                          recursion = FALSE;
                          defcall = TRUE;
                      check(inst.VidlInstance->body);
                          if(typeof(inst.VidlInstance->body.IDLclassCommon->exp_type)
                             != Kunknown)
                          {
                             defn.IDLclassCommon->deftype = inst.VidlInstance
                                                       ->body.IDLclassCommon->exp_type;
                             if(recursion) type_recursion(defn,
                                                       inst.VidlInstance->body);
                          }
                       }
            }
            foreachinSETInstance(defn.IDLclassCommon->overload,Sin,inst)
               if(typeof(inst.VidlInstance->body.IDLclassCommon->exp_type)
                  == Kunknown)
               {
                       type_recursion(defn,inst.VidlInstance->body);
                  inst.VidlInstance->body.IDLclassCommon->exp_type
                               = defn.IDLclassCommon->deftype;
               }

    }


/*          type cyclic definitions             */
/*+++++ NOT YET IMPLEMENTED ++++++*/
/* for now, remove all cyclic definitions */
foreachinSETdefinition(SP.Vstructure->defStore,Sdef,def)
    if(typeof(def) == Kcyclic)
            removeSETdefinition(SP.Vstructure->defStore,def);

/* remove definitions from assertion list - they are now stored in defStore */
foreachinSEQassertionStatement(SP.Vstructure->assertions,Sas,as)
    if(typeof(as) != Kassertion)
            removeSEQassertionStatement(SP.Vstructure->assertions,as);


/* check that all instances of the same definition have the same type */
    check_instance_types(SP);

    /* type bodies of assertions                    */
    defcall = FALSE;
    foreachinSEQassertionStatement(SP.Vstructure->assertions,Sas,as)
```

```
        if(typeof(as) == Kassertion)
            check(as.Vassertion->body);


}
else /* invariant - just clear assertions */
        SP.Vstructure->assertions = NULL;
}

/* analyze processes                                    */
/* parallels analysis of structures                     */
else if(typeof(SP) == Kprocess)
{

   /* collect definitions into defStore     */
   add_definitions(SP);

/* collect all definitions that a definition calls */
/* used to check that a noncyclic definition is, in fact, noncyclic          */
/* and to order noncyclic definitions for typing */

findcalls(SP);

/* mark all definitions as part of a cycle or not */

findcycles(SP);

/* check noncyclic definitions - if in a cycle change to
        a cyclic definition and issue a message to that effect   */

foreachinSETdefinition(SP.Vprocess->defStore,Sdef,def)
        if(typeof(def) == Knoncyclic && def.IDLclassCommon->cycle)
        {
            Warning1(800,def.IDLclassCommon->sourceposition,
                            def.IDLclassCommon->name->name);
            /* copy noncyclic attributes to a new cyclic definition */
            defname = malloc(strlen(def.IDLclassCommon->name->name)+1);
            strcpy(defname,def.IDLclassCommon->name->name);
            Sin = def.IDLclassCommon->overload;
            def.Vcyclic = Ncyclic;
            def.Vcyclic->name = NnameToken;
            def.Vcyclic->name->name = malloc(strlen(defname)+1);
            strcpy(def.Vcyclic->name->name,defname);
            def.Vcyclic->overload = Sin;
            Sdef->value = def;
        }

/* reverse topological sort of noncyclic definitions */
topsort(SP);

   /* make pass through noncyclic definitions               */
foreachinSETdefinition(SP.Vprocess->defStore,Sdefn,defn)
        defn.IDLclassCommon->visit = FALSE;

num_noncyclic = number_noncyclic(SP.Vstructure->defStore);
for(p = 1;p <= num_noncyclic;p++)
{
        defn = mindef(SP.Vprocess->defStore);
            /* analyze bodies of definition instances           */
       foreachinSETInstance(defn.IDLclassCommon->overload,Sin,inst)
       {
                if(typeof(inst) == KidlInstance)
                {
                    defcall = TRUE;
                    recursion = FALSE;
              check(inst.VidlInstance->body);
                    if(typeof(inst.VidlInstance->body.IDLclassCommon->exp_type)
                              != Kunknown)
                           {
                              defn.IDLclassCommon->deftype = inst.VidlInstance
                                                ->body.IDLclassCommon->exp_type;
                              if(recursion) type_recursion(defn,
                                                         inst.VidlInstance->body);
                           }
                        }
                 }
                 foreachinSETInstance(defn.IDLclassCommon->overload,Sin,inst)
                    if(typeof(inst.VidlInstance->body.IDLclassCommon->exp_type)
                       == Kunknown)
                        {
                           type_recursion(defn,inst.VidlInstance->body);
                        inst.VidlInstance->body.IDLclassCommon->exp_type
                                 = defn.IDLclassCommon->deftype;
                        }
   }


/*          type cyclic defintions              */
/*+++++ NOT YET IMPLEMENTED +++++*/
/* for now, remove all cyclic definitions */
foreachinSETdefinition(SP.Vprocess->defStore,Sdef,def)
   if(typeof(def) == Kcyclic)
          removeSETdefinition(SP.Vprocess->defStore,def);

/* remove definitions from assertion list - they are now stored in defStore */
foreachinSEQassertionStatement(SP.Vprocess->assertions,Sas,as)
   if(typeof(as) != Kassertion)
          removeSEQassertionStatement(SP.Vprocess->assertions,as);


/* Check that all instances of a definition return the same type */
   check_instance_types(SP);

   /* type bodies of assertions                         */
   defcall = FALSE;
   foreachinSEQassertionStatement(SP.Vprocess->assertions,Sas,as)
          if(typeof(as) == Kassertion)
             check(as.Vassertion->body);

   }

          /* output symbol table */
}

          SymbolTable(stdout,symbolTable,TWOPASS);

} /* of main */


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          x_type                                                          *
* Function which types expressions.                                   *
*  Recursively calls itself as it types subexpressions            *
*  Returns a typed expression that is the semantic meaning of the *
*  given expression.                                                 *
*                                                                          *
*          Last Revised:        June 10, 1986                              *
*******************************************************************/

expression x_type(exp,SP,defcall,defn,inst,as)
expression                  exp;       /* expression to type */
SYMBOL                      SP;        /* structure or process */
BOOLEAN                              defcall;  /* true if typing a definition */
```

```
definition              defn;       /* definition being typed(if definition) */
Instance                inst;   /* instance whose body is being typed */
assertionStatement      as;         /* assertion or definition being typed */
{
            SEQexpressionPair   Sorif;
            expressionPair                ep;

            typeTree            left_type;  /* type binary left subexp  */
            typeTree            right_type;/* type binary right subexp */
            int                           boper;              /* binary operator        */
            typeTree            value();    /* returns type of object in
                                                             singleton collection      */
            typeTree            type_name();          /* returns user type with
                                                             a given name                    */
            typeTree            obj_type1;/* type of objects in
                                                                      collection */
            typeTree            obj_type2;/* type of objects in
                                                                      collection */
            typeTree            arg_type;  /* type argument of function */
            int                           uoper;              /* unary operator          */
            BOOLEAN                                 OK;                 /* is type OK?                     */

            expression          arg;                  /* application argument      */
            SEQexpression                 saveargs; /* application arguments     */
            SETdefinition                 defSet;             /*  iterators ...             */
            SETdefinition                 Sdef;
            definition          def;
            BOOLEAN                                 found;              /* has definition been found?*/
            BOOLEAN                                 found1;
            int                           argType;  /* another argument type      */
            SEQexpression                 Sexp;               /*  iterators ...             */
            expression          a;
            SETString           wider;                /* type to widen to    */
            SETString           widearg;  /* type to be widened*/

            SEQquantifier                 quantStk;
            SEQquantifier                 quantSeq;
            SEQquantifier                 Squant;
            quantifier          q;
            SEQformal                     formSeq;
            SEQformal                     Sform;
            formal                        f;
            SETdefinition                 defSt;
            SETInstance                   Sin;
            Instance            in;

            SETport                       Spt;                /* iterators ...             */
            port                          pt;

            int                           references;/* number of objects a single
                                                             name refers to       */
            BOOLEAN                                 form;               /* is expression a formal */
            BOOLEAN                                 contrl;             /* is expression a control */
            BOOLEAN                                 df;                 /* is expression a definition */
            BOOLEAN                                 te;                 /* is expression a type exp */
            String                        tempname;/* name of a name expression
                                                             or application       */
            int                           position;  /* sourceposition of assertion
                                                             or definiiton */


if(defcall)
    position = defn.IDLclassCommon->sourceposition;
else
    position = as.IDLclassCommon->sourceposition;
```

```
/* type each kind of expression                    */
switch (typeof(exp)) {

        case Kconditional:

                /* type subexpressions of the conditional... */
                check(exp.Vconditional->test);

                if(typeof(exp.Vconditional->test.IDLclassCommon->exp_type)
                  != Kbool)
                          Warning0(701,position);

                check(exp.Vconditional->then);
                check(exp.Vconditional->otherwise);

                /* type all OrIf expressions              */
                foreachinSEQexpressionPair(exp.Vconditional->orif,Sorif,ep)
                {
                   check(ep->test);
                   check(ep->then);

                   /* test subexp must be boolean          */
                   if(typeof(ep->test.IDLclassCommon->exp_type) != Kbool)
                          Warning0(701,position);
                   /* OrIf-then subexp must have same type as then subexp */
                   if( typeof(ep->then.IDLclassCommon->exp_type) !=
                          typeof(exp.Vconditional->then.IDLclassCommon->exp_type)
                          && typeof(ep->then.IDLclassCommon->exp_type) != Kunknown
                          && typeof(exp.Vconditional->then.IDLclassCommon
                                 ->exp_type) != Kunknown
                          && !(numeric(ep->then.IDLclassCommon->exp_type) &&
                              numeric(exp.Vconditional->then.IDLclassCommon
                                 ->exp_type)) )
                          {Warning0(702,position); K(c1);}
             }
                /* type of then subexp must be same as else subexp    */
                if( typeof(exp.Vconditional->then.IDLclassCommon->exp_type) !=
                  typeof(exp.Vconditional->otherwise.IDLclassCommon->exp_type)
                  && typeof(exp.Vconditional->then.IDLclassCommon->exp_type)
                          != Kunknown
                  && typeof(exp.Vconditional->otherwise.IDLclassCommon
                          ->exp_type) != Kunknown
                          && !(numeric(ep->then.IDLclassCommon->exp_type) &&
                              numeric(exp.Vconditional->then.IDLclassCommon
                                 ->exp_type)) )
                          {Warning0(702,position); K(c2);}

                /* set type of conditional */

                if(typeof(exp.Vconditional->then.IDLclassCommon->exp_type)
                  != Kunknown && typeof(exp.Vconditional->then) != Kempty)
                   exp.Vconditional->exp_type = exp.Vconditional->
                                               then.IDLclassCommon->exp_type;
                else if(typeof(exp.Vconditional->otherwise.IDLclassCommon
                          ->exp_type) != Kunknown &&
                          typeof(exp.Vconditional->otherwise) != Kempty)
                   exp.Vconditional->exp_type = exp.Vconditional->
                                               otherwise.IDLclassCommon->exp_type;
                else foreachinSEQexpressionPair(exp.Vconditional->orif,Sorif,ep)
                   {
                          if(typeof(ep->then.IDLclassCommon->exp_type)
                              != Kunknown && typeof(ep->then) != Kempty)
                              exp.Vconditional->exp_type = ep->
                                               then.IDLclassCommon->exp_type;
                   }
```

```
            break;


case Kquantifier:

            /* Keep a record of all quantifiers                  */
            if(typeof(SP) == Kstructure)
                appendrearSEQquantifier(SP.Vstructure->quantStack,
                                                    exp.Vquantifier);
            else
                appendrearSEQquantifier(SP.Vprocess->quantStack,
                                                    exp.Vquantifier);


            exp.Vquantifier->control->exp_type.Vsingleton =
                                                            Nsingleton;

            /* type set subexp and body                          */
            check(exp.Vquantifier->set);

            exp.Vquantifier->control->exp_type.Vsingleton
                        ->object_type = exp.Vquantifier->set.IDLclassCommon
                                        ->exp_type;

            /* set subexp must be a collection                   */
            if(typeof(exp.Vquantifier->set.IDLclassCommon->exp_type)
               != Ksingleton && typeof(exp.Vquantifier->set.
               IDLclassCommon->exp_type) != Karbitrary)
                        Warning0(703,position);


            check(exp.Vquantifier->body);

            /* body of quantifier must be type boolean */
            if(typeof(exp.Vquantifier->body.IDLclassCommon->exp_type)
               != Kbool)
                  Warning0(704,position);

            exp.Vquantifier->exp_type.Vbool = Nbool;

            /* remove quantifier from record of quantifiers      */
            if(typeof(SP) == Kstructure)
                removelastSEQquantifier(SP.Vstructure->quantStack);
            else
                removelastSEQquantifier(SP.Vprocess->quantStack);

            break;


case Kbinary:
            boper = typeof(exp.Vbinary->op);

            /* type left and right subexps                       */
            check(exp.Vbinary->left);
            check(exp.Vbinary->right);

            left_type = exp.Vbinary->left.IDLclassCommon->exp_type;
            right_type = exp.Vbinary->right.IDLclassCommon->exp_type;

            /* singleton collection may be operands of value operators,
              in which case type is type of object in collection    */

            if(!(boper == Ksame || boper == Ksubset || boper == KpropSubset
                        || boper == KTunion || boper == Kintersect))
            {
               if(typeof(left_type) == Ksingleton)
                        left_type = value(left_type);
```

```
        if(typeof(right_type) == Ksingleton)
                right_type = value(right_type);
      }

/*  Handle errors in types of operands for various operators... */

      if(boper == Kand || boper == Kor)
      {
         if((typeof(left_type) != Kbool &&
                typeof(left_type) != Kunknown) ||
            (typeof(right_type) != Kbool &&
                typeof(right_type) != Kunknown))
                   Warning0(705,position);

                exp.Vbinary->exp_type.Vbool = Nbool;
      }

      if(boper==Kless || boper==KlessEq || boper==Kgreater ||
         boper==KgrtrEq )
      {
                OK = TRUE;
                if((typeof(left_type) != KTint &&
                    typeof(left_type) != Krat &&
                    typeof(left_type) != Kstr &&
                    typeof(left_type) != Kunknown) ||
                   (typeof(right_type) != KTint &&
                    typeof(right_type) != Krat &&
                    typeof(right_type) != Kstr &&
                    typeof(right_type) != Kunknown))
                           OK = FALSE;

                if(!OK)
                    Warning0(706,position);
                else
                    if(typeof(left_type) == Kstr)
                           if(!(typeof(right_type) == Kstr))
                               Warning0(707,position);

                exp.Vbinary->exp_type.Vbool = Nbool;
      }

      if(boper == Kequal || boper == KnotEqual)
      {
         if(typeof(left_type) == Karbitrary &&
            typeof(right_type) == Karbitrary)
                Warning0(708,position);
         else
         if(numeric(left_type))
         {
                if(!numeric(right_type) &&
                           typeof(right_type) != Kunknown)
                   Warning0(709,position);
         }
         else
         if(numeric(right_type) && typeof(left_type) != Kunknown)
                Warning0(709,position);
         else
         if(typeof(left_type) != typeof(right_type) &&
            typeof(left_type) != Kunknown &&
            typeof(right_type) != Kunknown)
                Warning0(709,position);

         exp.Vbinary->exp_type.Vbool = Nbool;
      }

      if(boper==Ksame || boper==Ksubset || boper==KpropSubset)
```

```
{
            if((typeof(left_type) != Ksingleton
                        && typeof(left_type) != Karbitrary
                        && typeof(left_type) != Kunknown) ||
              (typeof(right_type) != Ksingleton
                        && typeof(right_type) != Karbitrary
                        && typeof(right_type) != Kunknown))
            Warning0(710,position);

            exp.Vbinary->exp_type.Vbool = Nbool;
}

if(boper==KTunion || boper==Kintersect)
{
            OK = TRUE;
            if((typeof(left_type) != Ksingleton
                        && typeof(left_type) != Karbitrary
                        && typeof(left_type) != Kunknown) ||
              (typeof(right_type) != Ksingleton
                        && typeof(right_type) != Karbitrary
                        && typeof(right_type) != Kunknown))
                        OK = FALSE;

            if(!OK)
                Warning0(710,position);
            else
            {
                if(typeof(left_type) == Ksingleton)
                        obj_type1 = left_type.Vsingleton->object_type;
                else if(typeof(left_type) == Karbitrary)
                            obj_type1
                                    = left_type.Varbitrary->object_type;
                else
                        obj_type1.Vunknown = Nunknown;

                if(typeof(right_type) == Ksingleton)
                        obj_type2 = right_type.Vsingleton->object_type;
                else if(typeof(right_type) == Karbitrary)
                            obj_type2
                                = right_type.Varbitrary->object_type;
                else
                        obj_type2.Vunknown = Nunknown;

                if(typeof(obj_type1) != Kunknown &&
                   typeof(obj_type2) != Kunknown &&
                   typeof(obj_type1) != Knotype &&
                   typeof(obj_type2) != Knotype)
                        if(typeof(obj_type1) != typeof(obj_type2))
                            Warning0(711,position);
            }

            exp.Vbinary->exp_type.Varbitrary = Narbitrary;
            if(typeof(left_type) == Ksingleton)
                exp.Vbinary->exp_type.Varbitrary->object_type
                                    = left_type.Vsingleton->object_type;
            else if(typeof(left_type) == Karbitrary)
            {
                if(typeof(left_type.Varbitrary->object_type)
                        != Knotype)
                          exp.Vbinary->exp_type.Varbitrary->object_type
                                    = left_type.Varbitrary->object_type;
                else
                          exp.Vbinary->exp_type.Varbitrary->object_type
                                    = right_type.Varbitrary->object_type;
            }
            else
                exp.Vbinary->exp_type.Varbitrary
```

```
                                        ->object_type.Vunknown = Nunknown;
                }

                if(boper == KinSet)
                {
                        OK = TRUE;
                        if(typeof(right_type) != Kset &&
                           typeof(right_type) != Kseq &&
                           typeof(right_type) != Karbitrary &&
                           typeof(right_type) != Kunknown)
                        {
                            OK == FALSE;
                            Warning0(712,position);
                        }
                        if(OK && typeof(right_type) == Kset)
                        {
                        if(typeof(left_type) !=
                                    typeof(right_type.Vset->component) &&
                           typeof(left_type) != Kunknown)
                             Warning0(713,position);
                        }
                        else if(OK && typeof(right_type) == Kseq)
                        {
                        if(typeof(left_type) !=
                                    typeof(right_type.Vseq->component) &&
                           typeof(left_type) != Kunknown)
                             Warning0(714,position);
                        }


                        exp.Vbinary->exp_type.Vbool = Nbool;
                }

                if(boper==Kplus || boper==Kminus ||
                   boper==Ktimes || boper==Kdivide)
                {
                        if((typeof(left_type) != KTint &&
                           typeof(left_type) != Krat &&
                           typeof(left_type) != Kunknown) ||
                           (typeof(right_type) != KTint &&
                           typeof(right_type) != Krat &&
                           typeof(right_type) != Kunknown))
                                    Warning0(715,position);

                        if(typeof(left_type) == Kunknown &&
                           typeof(right_type) == Kunknown)
                            exp.Vbinary->exp_type.Vunknown = Nunknown;
                        else
                            if(typeof(left_type) == Krat ||
                               typeof(right_type) == Krat)
                                exp.Vbinary->exp_type.Vrat = Nrat;
                            else exp.Vbinary->exp_type.VTint = NTint;
                }
                break;

        case Kunary:
                uoper = typeof(exp.Vunary->op);

                /* type body of unary exp                      */
                check(exp.Vunary->body);
                arg_type = exp.Vunary->body.IDLclassCommon->exp_type;

                /* singleton collection may be argument of value operator,
                   in which case type is type of object in collection      */
        if(typeof(arg_type) == Ksingleton)
                arg_type = value(arg_type);


      /* Check for error in operand type                              */
      if(uoper == KUnaryPlus || uoper == KUnaryMinus)
      {
                if((typeof(arg_type) != KTint ||
                   typeof(arg_type) != Krat) &&
                   typeof(arg_type) != Kunknown)
                    Warning0(716,position);
      }
      else if(typeof(arg_type) != Kbool &&
             typeof(arg_type) != Kunknown)
                  Warning0(717,position);

      exp.Vunary->exp_type =
                exp.Vunary->body.IDLclassCommon->exp_type;

      break;



case Kapplication:

      /* save arguments and then type them        */
      saveargs = exp.Vapplication->arguments;
      foreachinSEQexpression(saveargs,Sexp,a)
      {
        check(a);
        Sexp->value = a;
      }

      /* save application name */
      tempname = malloc(strlen(exp.Vapplication->name->name)+1);
      strcpy(tempname,exp.Vapplication->name->name);

   retrievefirstSEQexpression(saveargs,arg);

      found = FALSE;

      /* check if application is a supplied function          */
      if(streq(exp.Vapplication->name->name,MEMBERS))
      {
        found = TRUE;
        exp.Vmembers = Nmembers;
        exp.Vmembers->argument = arg;
        exp.Vmembers->exp_type.Varbitrary = Narbitrary;
        arg_type = arg.IDLclassCommon->exp_type;
        if(typeof(arg_type) == Ksingleton)
                arg_type = arg_type.Vsingleton->object_type;
        if(typeof(arg_type) == Kset)
                exp.Vmembers->exp_type.Varbitrary->object_type
                        = arg_type.Vset->component;
        else if(typeof(arg_type) == Kseq)
                exp.Vmembers->exp_type.Varbitrary->object_type
                        = arg_type.Vseq->component;
        else
        {
                exp.Vmembers->exp_type.Varbitrary->object_type.Vunknown
                        = Nunknown;
                Warning0(718,position);
        }
      }
      else

      if(streq(exp.Vapplication->name->name,HEAD))
```

```
{
    found = TRUE;
    exp.Vhead = Nhead;
    exp.Vhead->argument = arg;
    exp.Vhead->exp_type.Vsingleton = Nsingleton;
    if(typeof(arg.IDLclassCommon->exp_type) == Kseq)
            exp.Vhead->exp_type.Vsingleton->object_type
                    = arg.IDLclassCommon->exp_type.Vseq->component;
    else if(typeof(arg.IDLclassCommon->exp_type) == Ksingleton)
    {
            if(typeof(arg.IDLclassCommon->exp_type.Vsingleton
                    ->object_type) == Kseq)
                exp.Vhead->exp_type.Vsingleton->object_type
                    = arg.IDLclassCommon->exp_type.Vsingleton
                            ->object_type.Vseq->component;
            else
                {
                    exp.Vhead->exp_type.Vsingleton->object_type.
                            Vunknown = Nunknown;
                    Warning0(719,position);
                }
    }
    else
    {
            exp.Vhead->exp_type.Vsingleton->object_type.Vunknown
                    = Nunknown;
            Warning0(719,position);
    }
}
else

if(streq(exp.Vapplication->name->name,TYPE))
{
    found = TRUE;
    exp.Vtype = Ntype;
    exp.Vtype->argument = arg;
    exp.Vtype->exp_type.Varbitrary = Narbitrary;
    if(typeof(arg.IDLclassCommon->exp_type) == Ksingleton)
            exp.Vtype->exp_type.Varbitrary->object_type
                    = arg.IDLclassCommon->exp_type.
                            Vsingleton->object_type;
    else if(typeof(arg.IDLclassCommon->exp_type) == Karbitrary)
            exp.Vtype->exp_type.Varbitrary->object_type
                    = arg.IDLclassCommon->exp_type.
                            Varbitrary->object_type;
    else
    {
            exp.Vtype->exp_type.Varbitrary->object_type.Vunknown
                    = Nunknown;
            Warning0(720,position);
    }
}
else

if(streq(exp.Vapplication->name->name,SIZE))
{
    found = TRUE;
    exp.Vsize = Nsize;
    exp.Vsize->argument = arg;
    exp.Vsize->exp_type.VTint = NTint;
    if(!(
            typeof(arg.IDLclassCommon->exp_type) == Kstr ||
            typeof(arg.IDLclassCommon->exp_type) == Kset ||
            typeof(arg.IDLclassCommon->exp_type) == Kseq ||
            typeof(arg.IDLclassCommon->exp_type) == Ksingleton ||
            typeof(arg.IDLclassCommon->exp_type) == Karbitrary))
                Warning0(721,position);
    if(typeof(arg.IDLclassCommon->exp_type) == Ksingleton)
            if(!(
                typeof(arg.IDLclassCommon->exp_type.Vsingleton
                                ->object_type) == Kstr ||
                typeof(arg.IDLclassCommon->exp_type.Vsingleton
                                ->object_type) == Kset ||
                typeof(arg.IDLclassCommon->exp_type.Vsingleton
                                ->object_type) == Kseq))
                    Warning0(721,position);
}
else
if(streq(exp.Vapplication->name->name,TAIL))
{
    found = TRUE;
    exp.Vtail = Ntail;
    exp.Vtail->argument = arg;
    exp.Vtail->exp_type.Vseq = Nseq;
    if(typeof(arg.IDLclassCommon->exp_type) == Kseq)
            exp.Vtail->exp_type.Vseq->component
                        = arg.IDLclassCommon->exp_type.Vseq->component;
    else if(typeof(arg.IDLclassCommon->exp_type) == Ksingleton)
            if(typeof(arg.IDLclassCommon->exp_type.Vsingleton
                        ->object_type) == Kseq)
                exp.Vtail->exp_type.Vseq->component
                        = arg.IDLclassCommon->exp_type.Vsingleton
                                ->object_type.Vseq->component;
    else
    {
            exp.Vtail->exp_type.Vseq->component.Vunknown
                        = Nunknown;
            Warning0(722,position);
    }
}

/* if in a definition, check for recursive call */
if(!found && defcall)
{
    if(typeof(defn) == Knoncyclic)
            if(streq(tempname,defn.Vnoncyclic->name->name))
            {
                recursion = TRUE;
                found = TRUE;
                exp.VdefnRef = NdefnRef;
                exp.VdefnRef->name = malloc(strlen(tempname)+1);
                strcpy(exp.VdefnRef->name,tempname);
                exp.VdefnRef->arguments = saveargs;
                exp.VdefnRef->reference.Vdefinition = defn;
                exp.VdefnRef->exp_type.Vunknown = Nunknown;
            }
}



/* if not a supplied function,
   check if an existing definition applies              */
if(!found)
{
            if(typeof(SP) == Kstructure)
                defSet = SP.Vstructure->defStore;
            else
                defSet = SP.Vprocess->defStore;

            /* look for declared definition with same name  */
            foreachinSETdefinition(defSet,Sdef,def)
            {
                found1 = FALSE;
```

```
                    if(streq(tempname,def.IDLclassCommon->name->name))
                    {
                      found = TRUE;
                      found1 = TRUE;
                      exp.VdefnRef = NdefnRef;
                      exp.VdefnRef->name = malloc(strlen(tempname)+1);
                      strcpy(exp.VdefnRef->name,tempname);
                      exp.VdefnRef->arguments = saveargs;
                      exp.VdefnRef->reference.Vdefinition = def;
                      exp.VdefnRef->exp_type
                                  = def.IDLclassCommon->deftype;
                    }

                    /* if defn found, check argument types  */
                    if(found1)
                    {
                    if(!check_arg_types(def,exp.VdefnRef->arguments))
                       Warning1(723,position,tempname);
                    }

                 }
           }

           /* if not definition, check if type widening */
           if(!found)
           {
              if(typeof(SP) == Kstructure)
              {
                     arg_type = type_name(exp.Vapplication->name->name,
                                                   SP.Vstructure);
                     if(type_in_structure(arg_type,SP.Vstructure))
                     {
                        found = TRUE;
                        reach(&wider,arg_type,TRUE);
                        reach(&widearg,typeof(arg.IDLclassCommon->exp_type,
                                         TRUE);
                        if(include(wider,widearg))
                           exp.Vapplication->exp_type = arg_type;
                        else
                              Warning0(732,position);
                     }
              }
              else /* process */
              /* Widening not yet supported in processes.
                 Parser must be modified to allow colons
                 in applications. */
                 ;
           }


           /* if nothing matches ...  */
           if (!found)
           {
              Warning1(724,position,exp.Vapplication->name->name);
              exp.Vapplication->exp_type.Vunknown = Nunknown;
           }

           break;

case Kdotted:
           check(exp.Vdotted->left);
           dot_type(exp,position);
           break;


case KtypeExpression:
           exp.VtypeExpression->exp_type.Varbitrary = Narbitrary;
```

```
        exp.VtypeExpression->exp_type.Varbitrary->object_type
                = exp.VtypeExpression->type;

        if(typeof(SP) == Kstructure)
        {
           /* Parser will not return a type expression unless the
           port is specified, so we know there must be an error */
           if(typeof(exp.VtypeExpression->portName) != KTvoid)
                   Warning0(725,position);
        }
        else /* a process port expression */
        {
           found = FALSE;
           foreachinSETport(SP.Vprocess->ports,Spt,pt)
                   if(streq(pt.IDLclassCommon->name,exp.VtypeExpression
                           ->portName.VnameToken->name))
                       found = TRUE;

           if(!found)
                   Warning0(726,position);
           else /* check port to see if it contains type */
           {
                   found = FALSE;
                   foreachinSETport(SP.Vprocess->ports,Spt,pt)
                   {
                       if(streq(exp.VtypeExpression->portName.VnameToken
                                       ->name,pt.IDLclassCommon->name))
                               if(type_in_structure(exp.VtypeExpression->
                                  type,pt.IDLclassCommon->data))
                                   found = TRUE;
                   }
                   if(found = FALSE)
                      Warning0(727,position);
           }
        }

        break;


case KnameExpr:

        tempname = malloc(strlen(exp.VnameExpr->name)+1);
        strcpy(tempname,exp.VnameExpr->name);
        saveargs = NULL;
        found = FALSE;

        form = FALSE;
        contrl = FALSE;
        df = FALSE;
        te = FALSE;

        /* if within a definition, check if name is a formal param */
        if(defcall == TRUE)
     foreachinSEQformal(inst.IDLclassCommon->formals,Sform,f)
     {
           if(streq(f->name->name,tempname))
           {
              exp.VformArg = NformArg;
                   exp.VformArg->name = malloc(strlen(tempname)+1);
                   strcpy(exp.VformArg->name,tempname);
              exp.VformArg->actual = f;
              exp.VformArg->exp_type.Vsingleton = Nsingleton;
              exp.VformArg->exp_type.Vsingleton->object_type
                           = f->type;
              form = TRUE;
                   found = TRUE;
           }
```

```
}

    /* check if control var for quantifier */
    if(!found)
    {

    if(typeof(SP) == Kstructure)
        quantStk = SP.Vstructure->quantStack;
    else
        quantStk = SP.Vprocess->quantStack;

    foreachinSEQquantifier(quantStk,Squant,q)
    {
        if(streq(tempname,q->control->name))
        {
                exp.Vcontrol = Ncontrol;
                exp.Vcontrol->name = malloc(strlen(tempname)+1);
                strcpy(exp.Vcontrol->name,tempname);
                exp.Vcontrol->owner = q;
                exp.Vcontrol->exp_type.Vsingleton = Nsingleton;
                if(typeof(q->set.IDLclassCommon->exp_type)== Karbitrary)
                    exp.Vcontrol->exp_type.Vsingleton->object_type
                            = q->set.IDLclassCommon->exp_type.Varbitrary
                                    ->object_type;
                else
                    exp.Vcontrol->exp_type.Vsingleton
                            ->object_type.Vunknown = Nunknown;

                contrl = TRUE;
                found = TRUE;
        }
    }

    }


    /* if in a definition, check for recursive call */
    if(!found && defcall)
    {
        if(typeof(defn) == Knoncyclic)
                if(streq(tempname,defn.Vnoncyclic->name->name))
                {
                    recursion = TRUE;
                    found = TRUE;
                    df = TRUE;
                    exp.VdefnRef = NdefnRef;
                    exp.VdefnRef->name = malloc(strlen(tempname)+1);
                    strcpy(exp.VdefnRef->name,tempname);
                    exp.VdefnRef->arguments = saveargs;
                    exp.VdefnRef->reference.Vdefinition = defn;
                    exp.VdefnRef->exp_type.Vunknown = Nunknown;
                }
    }


    /* check for a parameter-less definition */
    if(!found)
    {
    if(typeof(SP) == Kstructure)
        defSt = SP.Vstructure->defStore;
    else
        defSt = SP.Vprocess->defStore;


    foreachinSETdefinition(defSt,Sdef,def)
```

```
            {
                if(streq(def.IDLclassCommon->name->name,tempname))
                {
                  df = TRUE;
                  exp.VnameExpr->exp_type.Vunknown = Nunknown;
                  foreachinSETInstance(def.IDLclassCommon->overload,Sin,in)
                          if(len((pGenList)(in.IDLclassCommon->formals))==0)
                          {
                                  exp.VdefnRef = NdefnRef;
                                  exp.VdefnRef->name = malloc(strlen(tempname)+1);
                                  strcpy(exp.VdefnRef->name,tempname);
                                  exp.VdefnRef->arguments = saveargs;
                                  exp.VdefnRef->reference.VInstance = in;
                                  exp.VdefnRef->exp_type
                                          = def.IDLclassCommon->deftype;
                          }
                }
            }
            }

            /* name must refer to a type expression      if not found above */
            if(!form && !contrl && !df)
            {
               if(typeof(SP) == Kprocess)
               {
                       Warning0(728,position);
                       exp.VnameExpr->exp_type.Varbitrary = Narbitrary;
                       exp.VnameExpr->exp_type.Varbitrary
                               ->object_type.Vunknown = Nunknown;
               }
               else /* look for user type in structure */
               {
                       arg_type = type_name(tempname,SP.Vstructure);
                       if(type_in_structure(arg_type,SP.Vstructure))
                       {
                          te = TRUE;
                          exp.VtypeExpression = NtypeExpression;
                          exp.VtypeExpression->portName.VTvoid = NTvoid;
                          exp.VtypeExpression->type = arg_type;
                          exp.VtypeExpression->exp_type.Varbitrary
                                          = Narbitrary;
                          exp.VtypeExpression->exp_type.Varbitrary
                                  ->object_type = arg_type;
                       }
                       else
                       {
                          exp.VnameExpr->exp_type.Vunknown = Nunknown;
                          Warning0(729,position);
                       }
               }
            }


            break;


    case Kroot:
            exp.Vroot->exp_type.Vsingleton = Nsingleton;

            if(typeof(exp.Vroot->portName) == KTvoid)
            {
               if(typeof(SP) == Kstructure)
                       exp.Vroot->exp_type.Vsingleton->object_type
                               = type_name(SP.Vstructure->root.
                                       IDLclassCommon->name,SP.Vstructure);
               else /* in a process without a port specified */
```

```
                        Warning0(731,position);
                }
                else  /* port is specified */
                {
                        if(typeof(SP) == Kstructure)
                        {
                                Warning0(725,position);
                        }
                        else /* within a process */
                        {
                           found = FALSE;
                           foreachinSETport(SP.Vprocess->ports,Spt,pt)
                           {
                                if(streq(pt.IDLclassCommon->name,
                                        exp.Vroot->portName.VnameToken->name))
                                {
                                   found = TRUE;
                                   exp.Vroot->exp_type.Vsingleton->object_type
                                           = type_name(pt.IDLclassCommon->data
                                             ->root.IDLclassCommon->name,
                                             pt.IDLclassCommon->data);
                                }
                           }
                           if(!found)
                                Warning0(726,position);
                        }
              }
                break;

        case Kempty:
                exp.Vempty->exp_type.Varbitrary = Narbitrary;
                exp.Vempty->exp_type.Varbitrary->object_type.Vnotype
                        = Nnotype;
                break;


        case Ktrue:
                exp.Vtrue->exp_type.Vbool = Nbool;
                break;

        case Kfalse:
                exp.Vfalse->exp_type.Vbool = Nbool;
                break;

        case KintegerToken:
                exp.VintegerToken->exp_type.VTint = NTint;
                break;

        case KrationalToken:
                exp.VrationalToken->exp_type.Vrat = Nrat;
                break;

        case KstringToken:
                exp.VstringToken->exp_type.Vstr = Nstr;
                break;

        default:
                break;
        } /* end of switch */

        return(exp);
} /* End of x_type */
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*                                                                      *
*  Title:  Semantic Analysis of Assertions                  *
*  Filename:        ~kickenso/softlab/seman/symbol2.c                          *
*  Author:          Jerry Kickenson <kickenso@UNC>                              *
*                   Department of Computer Science                              *
*                   University of North Carolina                   *
*                   Chapel Hill, NC  27514                                      *
*                                                                      *
*  Copyright (C) The University of North Carolina, 1985      *
*                                                                      *
*  All rights reserved. No part of this software may be sold or   *
*  distributed in any form or by any means without the prior written      *
*  permission of the SoftLab Software Distribution Coordinator. *
*                                                                      *
*  Report problems to          softlab@unc (csnet) or                   *
*                              softlab!unc@CSNET-RELAY (ARPAnet)        *
*  Direct all inquiries to the SoftLab Software Distribution      *
*          Coordinator, at the above addresses.                         *
*                                                                      *
*  Function: routines used by the type checking procedure      *
*                                                                      *
*                                                                      *
****************************************************************** */


#include <stdio.h>
#include "SemanticAssert.h"
#include "macros.h"


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          dot_type   types dot expressions                  *
*                                                                      *
*          Last revised:        May 25, 1986                            *
*********************************************************************/

dot_type(x,position)
expression x;          /* dot expression to type */
int                    position;  /* sourceposition of assertion or definition
                                          in which dot expression appears */
{
   NonTerminal                nt;         /* nonterminal that is type of
                                             left expression */
   BOOLEAN                    TypeOK;  /* is type a user defined type? */
   BOOLEAN                    found;      /* has attribute been found? */
   SEQSYMBOL                  Ssy;
   SYMBOL                     sy;


   TypeOK = FALSE;


   if(typeof(x.Vdotted->left.IDLclassCommon->exp_type) == Ksingleton)
   {
          if(typeof(x.Vdotted->left.IDLclassCommon->exp_type.Vsingleton
                    ->object_type) == Kuser)
          {
              nt = x.Vdotted->left.IDLclassCommon->exp_type.Vsingleton
                    ->object_type.Vuser->NT;
              TypeOK = TRUE;
              x.Vdotted->exp_type.Vsingleton = Nsingleton;
          }
   }
   else if(typeof(x.Vdotted->left.IDLclassCommon->exp_type) == Karbitrary)
   {
          if(typeof(x.Vdotted->left.IDLclassCommon->exp_type.Varbitrary
                        ->object_type) == Kuser)
            {
              nt = x.Vdotted->left.IDLclassCommon->exp_type.Varbitrary
                        ->object_type.Vuser->NT;
              TypeOK = TRUE;
              x.Vdotted->exp_type.Varbitrary = Narbitrary;
            }
      }

      if(TypeOK)
      {
            if(typeof(nt) == KNodeNT || typeof(nt) == KClassNT)
            /* check all attributes of class and node nonterminals for
              a match with specified attribute                    */
            {
              if(typeof(nt.IDLclassCommon->locals) == Kscope)
              {
              found = FALSE;
              foreachinSEQSYMBOL(nt.IDLclassCommon->locals.Vscope->symbols,Ssy,sy)
              {
                    if(typeof(sy) == KAttribute)
                      if(streq(sy.VAttribute->name,x.Vdotted->right->name))
                      {
                            found = TRUE;
                            if(typeof(x.Vdotted->exp_type) == Ksingleton)
                              x.Vdotted->exp_type.Vsingleton->object_type
                                    = sy.VAttribute->type;
                            else
                              x.Vdotted->exp_type.Varbitrary->object_type
                                    = sy.VAttribute->type;
                      }
              }
              }
              if(!found)
              {
                Warning0(750,position);
                    x.Vdotted->exp_type.Varbitrary = Narbitrary;
                    x.Vdotted->exp_type.Varbitrary->object_type.Vunknown = Nunknown;
              }
            }
            else
            {
              Warning0(750,position);
              x.Vdotted->exp_type.Varbitrary = Narbitrary;
              x.Vdotted->exp_type.Varbitrary->object_type.Vunknown = Nunknown;
            }
      }
      else
      {
            x.Vdotted->exp_type.Varbitrary = Narbitrary;
            x.Vdotted->exp_type.Varbitrary->object_type.Vunknown = Nunknown;
            Warning0(751,position);
      }

}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *          reach       place in the given set strings representing all types    *
 *                      reachable from the given type.  Used to expand class     *
 *                      types into their constituent node types.                 *
 *                                                                                   *
 *          Last revised:          April 14, 1986                                    *
 *********************************************************************/

reach(t,incl,newset)
```

```
typeTree   t;           /* type that does the reaching */
SETString *incl;        /* set in which to place type representations */
BOOLEAN                         newset;    /* TRUE if new set, FALSE if add to set   */
{

   SEQNT            Snt;
   NT               nt;
   SETString        nt_reach();/* reach for nonterminals */

   if(newset)
      *incl = NULL;

   switch (typeof(t)) {
           case Kbool:
              addSETString(*incl,BOOL);
              break;

           case KTint:
              addSETString(*incl,INT);
              break;

           case Krat:
              addSETString(*incl,RAT);
              break;

           case Kstr:
              addSETString(*incl,STR);
              break;

           case Kset:
              addSETString(*incl,SET);
              break;

           case Kseq:
              addSETString(*incl,SEQ);
              break;

           case Kuser:
              if(typeof(t.Vuser->NT) != KClassNT)
                      {addSETString(*incl,t.Vuser->NT.IDLclassCommon->name);}
                     else
                     /* class type - trace down nonterminal descendants */
              foreachinSEQNT(t.Vuser->NT.VClassNT->descendants,Snt,nt)
                             *incl = nt_reach(nt,*incl);
              break;

           case Ksingleton:
              reach(t.Vsingleton->object_type,incl,1);
              break;

           case Karbitrary:
              break;

           }
}

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          nt_reach   places in the given set the string                  *
*                             representations of types reachable through the      *
*                             given nonterminal.                                  *
*                                                                                    *
*          Last revised:      April 14, 1986                                         *
*********************************************************************/

SETString nt_reach(n,i)
NT                 n;         /* nonterminal doing the reaching */
SETString i;          /* set in which to place type represetations */
```

```
{
        SETString strset;
        SEQNT           Snt;
        NT              nt;

        if(len( (pGenList)i) == 0)
           strset = NULL;
        else
           strset = i;

        if(typeof(n) != KClassNT)
          {addSETString(strset,n.IDLclassCommon->name);}
        else
          foreachinSEQNT(n.VClassNT->descendants,Snt,nt)
             strset = nt_reach(nt,strset);
        return(strset);
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       common          determines whether two sets of strings have*
*                       any elements in common                       *
*                                                                    *
*       Last revised:   April 19, 1986                               *
**********************************************************************/

common(s1,s2)
SETString s1,s2;
{

        BOOLEAN                 com;    /* result */
        SETString Sst;
        String          st;

        com = FALSE;
        foreachinSETString(s2,Sst,st)
           if(inSETString(s1,st))
                com = TRUE;

        return(com);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       include         determines whether the set of strings s1    *
*                       includes the set s2                        *
*                                                                    *
*       Last revised:   April 19, 1986                               *
**********************************************************************/

include(s1,s2)
SETString s1,s2;
{

        BOOLEAN                 com;
        SETString Sst;
        String          st;

        com = TRUE;

        foreachinSETString(s2,Sst,st)
           if(!inSETString(s1,st))
                com = FALSE;

        return(com);
}
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         check_arg_types              checks types of a list of arguments      *
*                                      against the formal types of the   *
*                                      declared declaration              *
*                                                                            *
*         Last revised:       April 19, 1986                                 *
*******************************************************************/

check_arg_types(defn,arglist)
definition  defn;       /* definition declared */
SEQexpression        arglist;    /* list of arguments */
{

          BOOLEAN                        cont;        /* continue */
          SETInstance          Sin;
          Instance    in;
          int                  n;          /* used by called routine */
          int                  i;
          formal               arg_f;      /* formal argument */
          expression arg_ac; /* actual argument */
          SETString inc1;
          SETString inc2;
          SETString Ss;
          String               s;

          n = 1;
          cont = TRUE;

          /* first check if there is an instance for which the arguments
             are not distinct by type; if so, then arguments check out */

          foreachinSETInstance(defn.IDLclassCommon->overload,Sin,in)
          {
             if(cont == TRUE)
                    if(!distinct_arg(in.IDLclassCommon->formals,arglist,n))
                       cont = FALSE;
          }

          /* if first check fails, check if the union of instance types
             for each argument includes the type of the actual argument */
          if(cont == TRUE)
          {
             for(i=1;i<=len((pGenList)arglist);i++)
             {
                    inc1 = NULL;
                    inc2 = NULL;

                    if(cont = TRUE)
                    {
                       foreachinSETInstance(defn.IDLclassCommon->overload,Sin,in)
                              if(len((pGenList)arglist) == len((pGenList) in.
                                        IDLclassCommon->formals))
                              {
                                 ithinSEQformal(in.IDLclassCommon->formals,i,arg_f);
                                 reach(arg_f->type,&inc1,0);
                              }
                       ithinSEQexpression(arglist,i,arg_ac);
                       reach(arg_ac.IDLclassCommon->exp_type,&inc2,1);

                       if(!include(inc1,inc2))
                              cont = FALSE;
                    }
             }
             if(cont == FALSE)
                    return(FALSE);       /* arguments still do not check out */
             else
                    return(TRUE);        /* arguments check out */
          }
          else
             return(TRUE);     /* arguments check out immediately */

}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         distinct_formal                    determine whether two sets of formal       *
*                                            arguments can be distinguished by length*
*                                            or by types                                    *
*                                                                                           *
*         Last revised:         April 19, 1986                                              *
*************************************************************************/

distinct_formal(form1,form2,i)
SEQformal              form1,form2;          /* formal arguments to check        */
int                    i;           /* number of formal being checked           */
{

          formal                arg1,arg2;
          SETString inc1,inc2;

          if(len((pGenList)form1) != len((pGenList)form2))
             return(TRUE);
          else
             if(i > len((pGenList)form1))
                    return(FALSE);
             else
             {
                    ithinSEQformal(form1,i,arg1);
                    ithinSEQformal(form2,i,arg2);
                    reach(arg1->type,&inc1,1);
                    reach(arg2->type,&inc2,1);
                    if(inSETString(inc1,UNKNOWN) || inSETString(inc2,UNKNOWN))
                    {
                       return(TRUE);
                    }
                    else
                    {
                       if(common(inc1,inc2))
                              return(distinct_formal(form1,form2,++i));
                       else
                              return(TRUE);
                    }
             }
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         distinct_arg          determine whether a set of formal arguments        *
*                               and a set of actual arguments can be               *
*                               distinguished by length or types          *
*                                                                                   *
*         Last revised:         April 19, 1986                                      *
*************************************************************************/

distinct_arg(form,args,i)
SEQformal              form;       /* formal arguments to check      */
SEQexpression          args;       /* actual arguments */
int                    i;          /* number of formal being checked           */
{

          formal                arg_formal;
          expression arg_actual;
          SETString inc1,inc2;
```

```
	if(len((pGenList)form) != len((pGenList)args))
	   return(TRUE);
	else
	   if(i > len((pGenList)form))
		return(FALSE);
	   else
	   {
		ithinSEQformal(form,i,arg_formal);
		ithinSEQexpression(args,i,arg_actual);
		reach(arg_formal->type,&inc1,1);
		reach(arg_actual.IDLclassCommon->exp_type,&inc2,1);
		if(inSETString(inc1,UNKNOWN) || inSETString(inc2,UNKNOWN))
		{
		   return(TRUE);
		}
		else
		{
		   if(include(inc1,inc2))
			return(distinct_arg(form,args,++i));
		   else
			return(TRUE);
		}
	   }
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*	len	return the length of a generic list		*
*								*
*	Last revised:	April 19, 1986				*
*********************************************************************/

len(list)
pGenList  list;
{

	int		length;
	pGenList  listptr;

	listptr = list;
	length = 0;

	while(listptr != NULL)
	{
	   length++;
	   listptr = listptr->next;
	}

	return(length);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*	type_name returns the user type with the given name	*
*								*
*	Last revised:	May 25, 1986				*
*********************************************************************/

typeTree  type_name(tname,St)
String		tname;	/* name of user type */
structure  St;		/* structure */
{

	typeTree   result;
	SEQSYMBOL	syms;	/* list of symbols in which to search for
				   non-terminal				*/
	SEQSYMBOL	Ssy;
	SYMBOL		sy;
```

```
        BOOLEAN                 found;     /* True if non-terminal is found */
        char            *malloc();

        result.Vuser = Nuser;
        result.Vuser->DT = NnameToken;
        result.Vuser->DT->name = malloc(strlen(tname)+1);
        strcpy(result.Vuser->DT->name,tname);

        if(typeof(St->locals) == Kscope)
           syms = St->locals.Vscope->symbols;

        found = FALSE;

        foreachinSEQSYMBOL(syms,Ssy,sy)
        {
           switch (typeof(sy))          {

                case KNodeNT:
                   if(streq(sy.IDLclassCommon->name,tname))
                   {
                        found = TRUE;
                        result.Vuser->NT.VNodeNT = sy.VNodeNT;
                   }
                   break;
                case KClassNT:
                   if(streq(sy.IDLclassCommon->name,tname))
                   {
                        found = TRUE;
                        result.Vuser->NT.VClassNT = sy.VClassNT;
                   }
                   break;
                case KPrivateNT:
                   if(streq(sy.IDLclassCommon->name,tname))
                   {
                        found = TRUE;
                        result.Vuser->NT.VPrivateNT = sy.VPrivateNT;
                   }
                   break;
                default:
                   /* no action */
                   break;
           }
        }


        return(result);

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       value     return the type of the object in a singleton collection   *
*                                                                            *
*       Last revised:       May 25, 1986                                     *
*********************************************************************/

typeTree  value(t)
typeTree  t;          /* type that is know to be a singleton collection */
{
   typeTree          result;

   if(typeof(t) == Ksingleton)
        result = t.Vsingleton->object_type;
   else
        result.Vunknown = Nunknown;

   return(result);
}
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*        type_in_structure    returns TRUE if the given type is found    *
*                                 in the given structure, FALSE otherwise   *
*                                                                           *
*        Last revised:        May 25, 1986                                  *
**********************************************************************/

type_in_structure(t,st)
typeTree  t;          /* type to find */
structure  st;        /* structure in which to look for type */
{
   BOOLEAN        result;
   SEQSYMBOL    Ssy;
   SYMBOL         sy;

   result = FALSE;

   if(typeof(st->locals) == Kscope && typeof(t) == Kuser)
          foreachinSEQSYMBOL(st->locals.Vscope->symbols,Ssy,sy)
          {
             if(is_nonterminal(sy) &&
                          streq(sy.IDLclassCommon->name,t.Vuser->DT->name))
                    result = TRUE;
          }

   return(result);

}

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*        is_nonterminal                returns TRUE if the given symbol is a    *
*                                      nonterminal, FALSE otherwise         *
*                                                                           *
*        Last revised:        May 25, 1986                                  *
**********************************************************************/

is_nonterminal(s)
SYMBOL s;          /* symbol to test */
{

   if(typeof(s) == KNodeNT || typeof(s) == KClassNT || typeof(s) == KPrivateNT)     return(TRUE);
   else
          return(FALSE);
}

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*        numeric             returns TRUE if the given type is integer or        *
*                            rational                                       *
*                                                                           *
*        Last revised:       May 31, 1986                                   *
**********************************************************************/

numeric(t)
typeTree  t;
{

   if(typeof(t) == KTint || typeof(t) == Krat)
          return(TRUE);
   else
          return(FALSE);
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*        type_recursion      type all recursive references of a definition *
*                            to itself                                      *
*                                                                           *
*          Last revised:          June 11, 1986                                                          *
***********************************************************************/

type_recursion(def,body)
definition  def;                /* definition that had a recursive call */
expression body;                /* body of recursive definition    */
{

    SEQexpressionPair           Sep;
    expressionPair     ep;
    SEQexpression      Sarg;
    expression                  arg;


    switch (typeof(body)) {

            case Kconditional:
                type_recursion(def,body.Vconditional->test);
                type_recursion(def,body.Vconditional->then);
                type_recursion(def,body.Vconditional->otherwise);
                foreachinSEQexpressionPair(body.Vconditional->orif,Sep,ep)
                {
                    type_recursion(def,ep->test);
                    type_recursion(def,ep->then);
                }
                break;

            case Kquantifier:
                type_recursion(def,body.Vquantifier->set);
                type_recursion(def,body.Vquantifier->body);
                break;

            case Kbinary:
                type_recursion(def,body.Vbinary->left);
                type_recursion(def,body.Vbinary->right);
                break;

            case Kunary:
                type_recursion(def,body.Vunary->body);
                break;

            case Kdotted:
                type_recursion(def,body.Vdotted->left);
                break;

            case KdefnRef:
                foreachinSEQexpression(body.Vapplication->arguments,Sarg,arg)
                         type_recursion(def,arg);

                if(streq(body.VdefnRef->name,def.IDLclassCommon->name->name))
                         body.VdefnRef->exp_type = def.IDLclassCommon->deftype;
                break;

            default:
                /* no other expressions can include a definition call */
                break;

            } /* end of switch */

}
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *                                                                       *
 *  Title:  Semantic Analysis of Assertions                        *
 *  Filename:       ~kickenso/softlab/seman/symbol3.c                         *
 *  Author:         Jerry Kickenson <kickenso@UNC>                                 *
 *                  Department of Computer Science                               *
 *                  University of North Carolina                          *
 *                  Chapel Hill, NC  27514                                       *
 *                                                                       *
 *  Copyright (C) The University of North Carolina, 1985           *
 *                                                                       *
 *  All rights reserved. No part of this software may be sold or   *
 *  distributed in any form or by any means without the prior written     *
 *  permission of the SoftLab Software Distribution Coordinator. *
 *                                                                       *
 *  Report problems to          softlab@unc (csnet) or                           *
 *                              softlab!unc@CSNET-RELAY (ARPAnet)         *
 *  Direct all inquiries to the SoftLab Software Distribution      *
 *          Coordinator, at the above addresses.                          *
 *                                                                       *
 *  Function: routines to handle definitions                       *
 *                                                                       *
 *                                                                       *
 ***************************************************************** */


 ************************************** **  Revision Log:
 *          $Log:    symbol3.c,v $
 * Revision 1.1  86/03/24  22:11:19  kickenso
 * Initial revision
 *
 *                                                                       *
 *  Edit Log:                                                            *
 *          May 25 1985 (kickenson)                                      *
 *                                                                       *
 ***************************************************************** */

#include <stdio.h>
#include "SemanticAssert.h"
#include "macros.h"


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *          add_definitions                                              *
 * collect all definitions together in the defStore set for each    *
 * structure and process;                                         *
 * while doing this, collect instances of an overloaded definition  *
 * together and check that such instances can be distinguished by        *
 * formal argument types                                          *
 *                                                                       *
 *          Last revised:          May 25 1986                           *
 ********************************************************************/

add_definitions(SP)
SYMBOL SP;          /* structure or process */
{

        SEQassertionStatement          Sas;
        assertionStatement    as;
        SEQassertionStatement          asserts;    /* lists of assertions */
        SETdefinition                  storedDefs;         /* list of defininitions */
        BOOLEAN                                new;            /* True if the instance is a
                                                          new definition      */
        BOOLEAN                                OK;             /* True if overloaded instances
                                                          are distinct                */
        Instance              newinst;  /* create a new instance */
        int                            n;
        SETdefinition                   Sdef;
        definition          def;
        SETInstance                     Sin;
        Instance            in;

        SEQformal                       Sf;
        formal                          f;


if(typeof(SP) == Kstructure)
        asserts = SP.Vstructure->assertions;
else
        asserts = SP.Vprocess->assertions;

storedDefs = NULL;

foreachinSEQassertionStatement(asserts,Sas,as)
{
        if(typeof(as) == Kcyclic)
        {
        new = TRUE;
        OK = TRUE;

        /* check to see if there is already a definition with the
           same name - if so check for distinguishable formal types */


        foreachinSETdefinition(storedDefs,Sdef,def)
           if(streq(as.Vcyclic->name->name,def.IDLclassCommon->name->name))
           {
             new = FALSE;
             foreachinSETInstance(def.IDLclassCommon->overload,Sin,in)
                     if(distinct_formal(as.Vcyclic->formals,
                                     in.IDLclassCommon->formals) != TRUE)
                     {
                             OK = FALSE;
                             Warning0(755,def.IDLclassCommon->sourceposition);
                     }

                 if(OK == TRUE)      /* add new instance to definition */
                 {
                     newinst.VidlInstance = NidlInstance;
                     newinst.VidlInstance->formals = as.Vcyclic->formals;
                     newinst.VidlInstance->body = as.Vcyclic->body;
                     addSETInstance(def.IDLclassCommon->overload,newinst);
              }
           }

           if(new == TRUE) /* add new definition to defStore */
           {
                   newinst.VidlInstance = NidlInstance;
                   newinst.VidlInstance->formals = as.Vcyclic->formals;
                   newinst.VidlInstance->body = as.Vcyclic->body;

                   addSETInstance(as.Vcyclic->overload,newinst);
                   addSETdefinition(storedDefs,as.Vcyclic);
           }

  }

        if(typeof(as) == Knoncyclic)
        {
        new = TRUE;
        OK = TRUE;

        /* check to see if there is already a definition with the
           same name - if so check for distinguishable formal types */
        foreachinSETdefinition(storedDefs,Sdef,def)
           if(streq(as.Vnoncyclic->name->name,def.IDLclassCommon->name->name))
           {
             new = FALSE;
             foreachinSETInstance(def.IDLclassCommon->overload,Sin,in)
                     if(distinct_formal(as.Vnoncyclic->formals,
                                        in.IDLclassCommon->formals) != TRUE)
                     {
                            OK = FALSE;
                            Warning0(755,def.IDLclassCommon->sourceposition);
                     }

                 if(OK == TRUE)     /* add new instance to definition */
                 {
                    newinst.VidlInstance = NidlInstance;
                    newinst.VidlInstance->formals = as.Vnoncyclic->formals;
                    newinst.VidlInstance->body = as.Vnoncyclic->body;
                    addSETInstance(def.IDLclassCommon->overload,newinst);
                    removeSEQassertionStatement(asserts,as);
             }
           }

           if(new == TRUE) /* add new definition to defStore */
           {
                 newinst.VidlInstance = NidlInstance;
                 newinst.VidlInstance->formals = as.Vnoncyclic->formals;
                 newinst.VidlInstance->body = as.Vnoncyclic->body;

                 addSETInstance(as.Vnoncyclic->overload,newinst);
                 addSETdefinition(storedDefs,as.Vnoncyclic);
           }

}


        if(typeof(as) == KprivateDefinition)
        {
        new = TRUE;
        OK = TRUE;

        /* check to see if there is already a definition with the
           same name - if so check for distinguishable formal types */

        foreachinSETdefinition(storedDefs,Sdef,def)
           if(streq(as.VprivateDefinition->name->name,
                           def.IDLclassCommon->name->name))
           {
             new = FALSE;
             foreachinSETInstance(def.IDLclassCommon->overload,Sin,in)
                     if(distinct_formal(as.VprivateDefinition->formals,
                            in.IDLclassCommon->formals) != TRUE)
                     {
                            OK = FALSE;
                            Warning0(755,def.IDLclassCommon->sourceposition);
                     }

                 if(OK == TRUE)     /* add new instance to definition */
                 {
                    newinst.VprivateInstance = NprivateInstance;
                    newinst.VprivateInstance->returnType
                            = as.VprivateDefinition->returnType;
                    addSETInstance(def.IDLclassCommon->overload,newinst);
             }
           }

           if(new == TRUE) /* add new definition to defStore */
           {
```

```
                    newinst.VprivateInstance = NprivateInstance;
                    newinst.VprivateInstance->returnType
                         = as.VprivateDefinition->returnType;

                    addSETInstance(as.VprivateDefinition->overload,newinst);
                    addSETdefinition(storedDefs,as.VprivateDefinition);
               }
          }
     }


          if(typeof(SP) == Kstructure)
             SP.Vstructure->defStore = storedDefs;
          else
             SP.Vprocess->defStore = storedDefs;
}



/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         check_instance_types checks that all instances of the same          *
*                                       definition return the same type        *
*                                                                              *
*         Last revised:         May 25, 1986                                   *
*****************************************************************************/

check_instance_types(SP)
SYMBOL SP;        /* structure or process */
{
   SETdefinition      DS;
   SETdefinition      Sdefn;
   definition                defn;
   SETInstance               Sin;
   Instance                  in;


   if(typeof(SP) == Kstructure)
          DS = SP.Vstructure->defStore;
   else
          DS = SP.Vprocess->defStore;


   foreachinSETdefinition(DS,Sdefn,defn)
   {
          if(typeof(defn) != KprivateDefinition)
             foreachinSETInstance(defn.IDLclassCommon->overload,Sin,in)
             {
                    if(typeof(in.VidlInstance->body.IDLclassCommon->exp_type)
                            != typeof(defn.IDLclassCommon->deftype))
                       Warning0(756,defn.IDLclassCommon->sourceposition);
                    else /* same type - check collections */
                    {
                       if(typeof(defn.IDLclassCommon->deftype) == Ksingleton)
                       {
                              if(typeof(defn.IDLclassCommon->deftype.Vsingleton
                                       ->object_type) !=
                                 typeof(in.VidlInstance->body.IDLclassCommon
                                       ->exp_type.Vsingleton->object_type))
                                  Warning0(757,defn.IDLclassCommon->sourceposition);
                       }
                       else if(typeof(defn.IDLclassCommon->deftype) == Karbitrary)
                       {
                              if(typeof(defn.IDLclassCommon->deftype.Varbitrary
                                       ->object_type) !=
                                 typeof(in.VidlInstance->body.IDLclassCommon
                                       ->exp_type.Varbitrary->object_type) &&
                              typeof(in.VidlInstance->body.IDLclassCommon
                                        ->exp_type.Varbitrary->object_type) !=
                                        Knotype)
                               Warning0(757,defn.IDLclassCommon->sourceposition);
                         }
                    }
             }
             else /* private definition */
             foreachinSETInstance(defn.IDLclassCommon->overload,Sin,in)
             {
                    if(typeof(in.VprivateInstance->returnType)
                              != typeof(defn.IDLclassCommon->deftype))
                        Warning0(756,defn.IDLclassCommon->sourceposition);
                    else /* same type - check collections */
                    {
                        if(typeof(defn.IDLclassCommon->deftype) == Ksingleton)
                        {
                              if(typeof(defn.IDLclassCommon->deftype.Vsingleton
                                        ->object_type) !=
                                 typeof(in.VprivateInstance->returnType.
                                        Vsingleton->object_type))
                               Warning0(757,defn.IDLclassCommon->sourceposition);
                        }
                        else if(typeof(defn.IDLclassCommon->deftype) == Karbitrary)
                        {
                              if(typeof(defn.IDLclassCommon->deftype.Varbitrary
                                        ->object_type) !=
                                 typeof(in.VprivateInstance->returnType.
                                        Varbitrary->object_type))
                               Warning0(757,defn.IDLclassCommon->sourceposition);
                        }
                    }
             }
   }
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          findcalls   Store with each definition the definitions it  *
*                                calls                                      *
*                                                                           *
*          Last revised:         June 10, 1986                              *
*********************************************************************/

findcalls(SP)
SYMBOL SP;        /* structure or process */
{

   SETdefinition     defs;      /* definitions stored with structure
                                            or process */
   SETdefinition     Sd;
   definition                   d;
   SETInstance                  Sin;
   Instance                     in;


   if(typeof(SP) == Kstructure)
          defs = SP.Vstructure->defStore;
   else
          defs = SP.Vprocess->defStore;

   foreachinSETdefinition(defs,Sd,d)
          foreachinSETInstance(d.IDLclassCommon->overload,Sin,in)
              if(typeof(in) == KidlInstance)
                 trackcalls(d,in.VidlInstance->body,defs);
}
```

```
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       trackcalls  track down all definition calls made in the  *
*                           given definition body.                    *
*                                                                        *
*       Last revised:       June 10, 1986                               *
********************************************************************/

trackcalls(def,body,storedDefs)
definition  def;                /* definition whose calls are to be recorded */
expression body;                /* body to track through              */
SETdefinition       storedDefs;         /* list of definitions in structure/process */
{

   SETdefinition    Sd;
   definition               d;
   SEQexpressionPair        Sep;
   expressionPair   ep;
   SEQexpression    Sarg;
   expression               arg;


   switch (typeof(body)) {

         case Kconditional:
            trackcalls(def,body.Vconditional->test,storedDefs);
            trackcalls(def,body.Vconditional->then,storedDefs);
            trackcalls(def,body.Vconditional->otherwise,storedDefs);
            foreachinSEQexpressionPair(body.Vconditional->orif,Sep,ep)
            {
               trackcalls(def,ep->test,storedDefs);
               trackcalls(def,ep->then,storedDefs);
            }
            break;

         case Kquantifier:
            trackcalls(def,body.Vquantifier->set,storedDefs);
            trackcalls(def,body.Vquantifier->body,storedDefs);
            break;

         case Kbinary:
            trackcalls(def,body.Vbinary->left,storedDefs);
            trackcalls(def,body.Vbinary->right,storedDefs);
            break;

         case Kunary:
            trackcalls(def,body.Vunary->body,storedDefs);
            break;

         case Kdotted:
            trackcalls(def,body.Vdotted->left,storedDefs);
            break;

         case KnameExpr:
            /* check if a definition */
            foreachinSETdefinition(storedDefs,Sd,d)
                  if(streq(body.VnameExpr->name,d.IDLclassCommon->name->name))
                     addSETdefinition(def.IDLclassCommon->calls,d);
            break;

         case Kapplication:
            foreachinSETdefinition(storedDefs,Sd,d)
                  if(streq(body.Vapplication->name->name,
                           d.IDLclassCommon->name->name))
                     addSETdefinition(def.IDLclassCommon->calls,d);
            foreachinSEQexpression(body.Vapplication->arguments,Sarg,arg)
                  trackcalls(def,arg,storedDefs);
            break;
            default:
                /* no other expressions can include a definition call */
                break;

            } /* end of switch */

}


int         cyclesize; /* number of definitions in a cycle, used to
                              distinguish cycles of only 1 def (recursion)
                              from real cyclic definitions          */
                             /* Global to findcycles and dfs_cycle  */

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           findcycles mark all definitions in the given structure or     *
*                             process that are part of a cycle             *
*                                                                                      *
*           Last revised:        June 12, 1985                                    *
*                                                                                      *
*           Algorithm adapted from Aho,Hopcroft,Ullman,                   *
*                             "Data Structures and Algorithms"              *
*********************************************************************/

findcycles(SP)
SYMBOL SP;
{

    SETdefinition       Sd;
    definition               d,df;
    SETdefinition       defs,Sdf;
    int                      order = 1;
    definition               maxindex();

    if(typeof(SP) == Kstructure)
            defs = SP.Vstructure->defStore;
    else
            defs = SP.Vprocess->defStore;

    /* order the definitions in reverse order */
    foreachinSETdefinition(defs,Sd,d)
    {
            if(!(d.IDLclassCommon->visit))
                dfs_topsort(d,&order);
    }

    /* reverse all call arcs in the graph */

    foreachinSETdefinition(defs,Sd,d)
            foreachinSETdefinition(d.IDLclassCommon->calls,Sdf,df)
                addSETdefinition(df.IDLclassCommon->revcalls,d);


    /* mark all definitions involved in a cycle */
    foreachinSETdefinition(defs,Sd,d)
            d.IDLclassCommon->visit = FALSE;

    while(left_tovisit(defs))
    {
            d = maxindex(defs);
        cyclesize = 0;
        dfs_cycle(d);
        if(cyclesize == 1)
                d.IDLclassCommon->cycle = FALSE;
    }
```

```
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           left_tovisit returns TRUE if the given set of definitions*
*                              has at least one unvisited member          *
*                                                                           *
*           Last revised:      June 12, 1986                               *
**********************************************************************/

left_tovisit(defs)
SETdefinition        defs;      /* definitions to look for unvisited def in */
{

   SETdefinition     Sd;
   definition              d;
   BOOLEAN                 result;

   result = FALSE;
   foreachinSETdefinition(defs,Sd,d)
          if(!(d.IDLclassCommon->visit))
             result = TRUE;

   return(result);
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           maxindex  return the definition in the given set of       *
*                              definitions with the highest index that is also     *
*                              unvisited.                                  *
*                                                                           *
*           Last revised:      June 12, 1986                               *
**********************************************************************/

definition  maxindex(defs)
SETdefinition        defs;      /* set of definitions to search */
{

   SETdefinition     Sd;
   definition              d;
   definition              result;
   int                     maxsofar = 0;

   foreachinSETdefinition(defs,Sd,d)
          if(d.IDLclassCommon->index > maxsofar &&
             !(d.IDLclassCommon->visit))
          {
             maxsofar = d.IDLclassCommon->index;
             result = d;
          }

   return(result);
}



/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           dfs_cycle  depth first search of a DAG modified to        *
*                              mark nodes that are in a cycle              *
*                                                                           *
*           Last revised:      June 12, 1985                               *
**********************************************************************/
```

```
dfs_cycle(def)
definition  def;                    /* definition to mark as cyclic or not */
{

   SETdefinition      Sd;
   definition                   d;

   def.IDLclassCommon->visit = TRUE;
   def.IDLclassCommon->cycle = TRUE;

   cyclesize++;

   foreachinSETdefinition(def.IDLclassCommon->revcalls,Sd,d)
           if(!(d.IDLclassCommon->visit))
              dfs_cycle(d,cyclesize);

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          topsort                 reverse topological sort of definitions in    *
*                                  given structure or process.  DAG arcs are   *
*                                  definition calls.                                    *
*                                                                                            *
*          Last revised:           June 10, 1986                                      *
*********************************************************************/

topsort(SP)
SYMBOL SP;       /* structure or process */
{

   SETdefinition      Sd;
   definition                   d;
   SETdefinition      defs;
   int                          order = 1; /* order of definition */

   if(typeof(SP) == Kstructure)
           defs = SP.Vstructure->defStore;
   else
           defs = SP.Vprocess->defStore;

   foreachinSETdefinition(defs,Sd,d)
           d.IDLclassCommon->visit = FALSE;

   foreachinSETdefinition(defs,Sd,d)
           if(typeof(d) == Knoncyclic)
              if(!(d.IDLclassCommon->visit))
                 dfs_topsort(d,&order);
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          dfs_topsort             depth first search modified to do a reverse  *
*                                  topological sort                                     *
*                                                                                            *
*          Last revised:           June 10, 1986                                      *
*********************************************************************/

dfs_topsort(def,order)
definition  def;        /* definition that is node of DAG */
int                     *order;    /* order of definition */
{

   SETdefinition      Sd;
   definition                   d;

   def.IDLclassCommon->visit = TRUE;
   foreachinSETdefinition(def.IDLclassCommon->calls,Sd,d)
```

```
            if(!(d.IDLclassCommon->visit))
               dfs_topsort(d,order);

   def.IDLclassCommon->index = (*order)++;
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           mindef              returns the noncyclic definition that has     *
*                               the lowest order number and has not yet been        *
*                               typed.                                                    *
*                                                                                         *
*           Last revised:       June 10, 1986                                             *
**********************************************************************/

definition  mindef(defs)
SETdefinition         defs;     /* definitions from which to find minimum */
{

   SETdefinition      Sd;
   definition                   d;
   definition                   result;
   int                          minsofar = 1000;

   foreachinSETdefinition(defs,Sd,d)
            if( (d.IDLclassCommon->index < minsofar) && typeof(d) == Knoncyclic
              && !(d.IDLclassCommon->visit) )
            {
               minsofar = d.IDLclassCommon->index;
               result = d;
            }

   result.IDLclassCommon->visit = TRUE;
   return(result);
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           number_noncyclic    returns the number of noncyclic           *
*                                             definitions in the given set      *
*                                                                                         *
*           Last revised:       June 10, 1986                                             *
**********************************************************************/

number_noncyclic(defs)
SETdefinition         defs;     /* set in which to count noncyclic definitions */
{

   int                result;
   SETdefinition      Sd;
   definition                   d;

   result = 0;
   foreachinSETdefinition(defs,Sd,d)
            if(typeof(d) == Knoncyclic)
               result++;

   return(result);
}
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *                                                                       *
 *  Title:  Postfix Code Generation for Assertion Checker          *
 *  Filename:      ~kickenso/softlab/codegen/codegen.c                  *
 *  Author:        Jerry Kickenson <kickenso@unc>                        *
 *                 Department of Computer Science                        *
 *                 University of North Carolina                         *
 *                 Chapel Hill, NC  27514                                *
 *                                                                       *
 *  Copyright (C) The University of North Carolina, 1985           *
 *                                                                       *
 *  All rights reserved. No part of this software may be sold or  *
 *  distributed in any form or by any means without the prior written  *
 *  permission of the SoftLab Software Distribution Coordinator. *
 *                                                                       *
 *  Report problems to         softlab@unc (csnet) or                    *
 *                             softlab!unc@CSNET-RELAY (ARPAnet)        *
 *  Direct all inquiries to the SoftLab Software Distribution      *
 *        Coordinator, at the above addresses.                          *
 *                                                                       *
 *  Function: main driver for code generation for assertion checker  *
 *                                                                       *
 *                                                                       *
 ******************************************************************* */

  $Header: gencode.c,v 1.7 85/12/19 12:47:15 kickenso Exp $

/* ******************************************************************** **  Revision Log:
 *        $Log:     gencode.c,v $
 * Revision 1.7  85/12/19  12:47:15  kickenso
 * This version works!  Exhaustive testing will follow.
 *
 * Revision 1.6  85/12/17  12:18:03  kickenso
 * handles all expressions - compiles
 * next: runtime checking
 *
 * Revision 1.5  85/12/15  13:19:30  kickenso
 * idlc version - still must add quantifiers & conditionals
 *
 * Revision 1.4  85/12/08  21:44:21  kickenso
 * pointer to postfix array now passed to gencode procedure
 *
 * Revision 1.3  85/12/08  15:21:04  kickenso
 * this version compiles!
 *
 * Revision 1.2  85/12/08  15:07:07  kickenso
 * created main driver

 * Revision 1.1  85/12/08  14:27:02  kickenso
 * Initial revision
 *
 *                                                                       *
 *  Edit Log:                                                            *
 *    December 8, 1985 (kickenson).                                     *
 *                                                                       *
 ******************************************************************* */


#include <stdio.h>
#include "GeneratePostfix.h"
#include "instructions.h"

#define gencode(BOD,PF)        generatecode(BOD,PF,in,SP)
#define streq(s1,s2)   strcmp(s1,s2) == 0


typedef int            Operation;
typedef char                    BYTE;
typedef SEQInteger   SEQOperation;
typedef int             boolean;


int                    level = 0;           /* level of a nested quantifier    */
SEQInteger             loc_stack = NULL;    /* stack of code array locations,
                                                      used by conditionals                  */
int                    currentpos = 0;              /* current position in code array */


main()
{
    scope                       st;         /* input symboltable  */
    SETString                               invs;       /*  iterators ...          */
    SEQSYMBOL                               SSYMBOL;
    SYMBOL                                  SP;
    SEQassertionStatement       Sas;
    assertionStatement          as;
    SETdefinition               Sdef;
    definition                              def;
    SETInstance                             Sin;
    Instance                                in;

    Operation                               ea;         /* endAssertion entry in code     */



    /* read in symbol table       */
    st = SymbolTable(stdin);


    invs = NULL;
    /* collect invariant structure names        */
    foreachinSEQSYMBOL(st->symbols,SSYMBOL,SP)
    {
            if(typeof(SP) == Kprocess)
                addSETString(invs,SP.Vprocess->invariant->name);
    }

    /* generate code for all assertions and definition instance bodies          */
    foreachinSEQSYMBOL(st->symbols,SSYMBOL,SP)
    {
            /* first handle structures, excepting invariant structures */
            if(typeof(SP) == Kstructure)
                if(!inSETString(invs,*SP.Vstructure->name))
                {
                        foreachinSEQassertionStatement(SP.Vstructure->assertions,
                                                            Sas,as)
                            if(typeof(as) == Kassertion)
                            {
                                    currentpos = 0;
                                    gencode(as.Vassertion->body,
                                            &(as.Vassertion->postfixBody));
                                    addtocode(ENDASSERTION,&(as.Vassertion->postfixBody));
                            }

                        foreachinSETdefinition(SP.Vstructure->defStore,Sdef,def)
                        {
                            if(typeof(def) == Kcyclic)
                                    foreachinSETInstance(def.Vcyclic->overload,Sin,in)
```

```
                                        {
                                            currentpos = 0;
                                            gencode(in.VidlInstance->body,
                                                    &(in.VidlInstance->postfixDefn));
                                            addtocode(RETURN,&(in.VidlInstance->postfixDefn));
                                        }
                                else if(typeof(def) == Knoncyclic)
                                            foreachinSETInstance(def.Vnoncyclic->overload,Sin,in)
                                            {
                                                    currentpos =0;
                                                    gencode(in.VidlInstance->body,
                                                                &(in.VidlInstance->postfixDefn));
                                                addtocode(RETURN,&(in.VidlInstance->postfixDefn));
                                            }
                            }
                    }

                /* now handle processes            */
                if(typeof(SP) == Kprocess)
                {
                            foreachinSEQassertionStatement(SP.Vprocess->assertions,
                                                                        Sas,as)
                                if(typeof(as) == Kassertion)
                                {
                                        currentpos = 0;
                                        gencode(as.Vassertion->body,
                                                    &(as.Vassertion->postfixBody));
                                        addtocode(ENDASSERTION,&(as.Vassertion->postfixBody));
                                }

                            foreachinSETdefinition(SP.Vprocess->defStore,Sdef,def)
                            {
                                if(typeof(def) == Kcyclic)
                                        foreachinSETInstance(def.Vcyclic->overload,Sin,in)
                                        {
                                            currentpos = 0;
                                            gencode(in.VidlInstance->body,
                                                    &(in.VidlInstance->postfixDefn));
                                            addtocode(RETURN,&(in.VidlInstance->postfixDefn));
                                        }
                                else if(typeof(def) == Knoncyclic)
                                         foreachinSETInstance(def.Vnoncyclic->overload,Sin,in)
                                            {
                                                    currentpos = 0;
                                                    gencode(in.VidlInstance->body,
                                                                &(in.VidlInstance->postfixDefn));
                                                addtocode(RETURN,&(in.VidlInstance->postfixDefn));
                                            }
                            }
                }
        }

        /* write out table with generated code included       */
        PostfixCode(stdout,st,TWOPASS);

} /* end of main */


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           generatecode            generates postfix code equivalent to the given          *
*                                   expression tree.  Method uses a recursive tree  *
*                                   traversal.  Resulting code is an array of one*
*                                   byte integers (C char).  Maximum size of array  *
*                                   1000.                                                              *
*                                                                                                      *
*                                                                                                      *
*          Last revised:          June 7, 1986                                              *
*********************************************************************/

generatecode(body,code_array,in,SP)
expression body;                              /* expression tree     */
SEQOperation      *code_array;                        /* generated code passed back up */
Instance   in;                                /* definition instance */
SYMBOL            SP;                                 /* structure/process   */
{

   Operation                    op;        /* appropriate operator */
   BYTE            hi;          /* high byte of reference index   */
   BYTE            lo;          /* low byte of reference index    */
   int                          index;     /* index into string, rational, etc. array */

   SEQexpression   Sarg;        /*  iterators ...       */
   expression                   arg;

   double          atof();

   int                          startpos;  /* start position in code array
                                                   of quantifier                 */
   int                          loc;                /* location to be saved                */
   int                          i;                  /*  iterators ...                */
   SEQexpressionPair            Sep;
   expressionPair  ep;

   int                          loc_count = 0;      /* saves locations for OrIf
                                                     conditionals                */
   int                          tipe;


   /* generate code for different types of expressions  */
   switch (typeof(body))     {

      case Kunary:

               gencode(body.Vunary->body,code_array);

               /* determine operator type */
               if(typeof(body.Vunary->op) == KUnaryPlus);
                       /* no action need be taken */
               if(typeof(body.Vunary->op) == KUnaryMinus)
                       op = unaryMinusOp;
               if(typeof(body.Vunary->op) == Knot)
                       op = notOp;

               if(typeof(body.Vunary->op) != KUnaryPlus)
               {
                       body.Vunary->valuepos = currentpos;
                  addtocode(op,code_array);
               }

               break;

          case Kbinary:

               gencode(body.Vbinary->left,code_array);
               gencode(body.Vbinary->right,code_array);

               if(typeof(body.Vbinary->right) != KformArg &&
                  typeof(body.Vbinary->right) != Kcontrol)
               {
                  if(typeof(body.Vbinary->right.IDLclassCommon->exp_type)
                     == Ksingleton)
                     tipe = typeof(body.Vbinary->right.IDLclassCommon
                  ->exp_type.Vsingleton->object_type);
        else
      tipe = typeof(body.Vbinary->right.IDLclassCommon->exp_type);
}
else
   tipe = typeof(body.Vbinary->right.IDLclassCommon->exp_type);


/* determine operator type  */
switch (typeof(body.Vbinary->op)) {

        case Kand:
            op = andOp;
            break;

        case Kor:
            op = orOp;
            break;

        case KTunion:
            op = unionOp;
            break;

        case Kintersect:
            op = intersectOp;
            break;

        case Kplus:
            op = plusOp;
            break;

        case Kminus:
            op = minusOp;
            break;

        case Ktimes:
            op = timesOp;
            break;

        case Kdivide:
            op = divideOp;
            break;

        case Kless:
            /* determine which type of < opeator */
            if(tipe == Kstr)
                    op = str_lessOp;
            else
                    op = num_lessOp;
            break;

        case KlessEq:
            /* determine which type of <= operator */
            if(tipe == Kstr)
                    op = str_lessEqOp;
            else
                    op = num_lessEqOp;
            break;

        case Kgreater:
            /* determine which type of > operator */
            if(tipe == Kstr)
                    op = str_greaterOp;
            else
                    op = num_greaterOp;
            break;
```

```
case KgrtrEq:
    /* determine which type of >= operator */
    if(tipe == Kstr)
            op = str_grtrEqOp;
    else
            op = num_grtrEqOp;
    break;

case Kequal:
    /* determine which type of equal operator */
    switch (tipe) {
            case Kbool:
                op = bool_equalOp;
                break;
            case KTint:
                op = num_equalOp;
                break;
            case Krat:
                op = num_equalOp;
                break;
            case Kstr:
                op = str_equalOp;
                break;
            case Kset:
                op = set_equalOp;
                break;
            case Kseq:
                op = seq_equalOp;
                break;
            case Ksingleton:
                op = node_equalOp;
                break;
            }
    break;

case KnotEqual:
    /* determine which type of ~= operator */
    switch (tipe) {
            case Kbool:
                op = bool_notEqualOp;
                break;
            case KTint:
                op = num_notEqualOp;
                break;
            case Krat:
                op = num_notEqualOp;
                break;
            case Kstr:
                op = str_notEqualOp;
                break;
            case Kset:
                op = set_notEqualOp;
                break;
            case Kseq:
                op = seq_notEqualOp;
                break;
            case Ksingleton:
                op = node_notEqualOp;
                break;
            }
    break;

case Ksame:
    op = sameOp;
    break;

case KinSet:
```

```
                    /* determine which kind of In operator */
                    switch (tipe) {
                            case Kset:
                                op = inSetOp;
                                break;
                            case Kseq:
                                op = inSeqOp;
                                break;
                            case Karbitrary:
                                op = inCollectionOp;
                                break;
                            }
                    break;

                case Ksubset:
                    op = subsetOp;
                    break;

                case KpropSubset:
                    op = propSubsetOp;
                    break;

                }

        body.Vbinary->valuepos = currentpos;
        addtocode(op,code_array);
        break;

    case Kdotted:

        gencode(body.Vdotted->left,code_array);

        op = dotOp;
        body.Vdotted->valuepos = currentpos;
        addtocode(op,code_array);

        /* calculate index into string reference array */
        index = add_String(SP,body.Vdotted->right->name);
        hi = (char) (index >> 8);
        lo = (char) index;
        addtocode(lo,code_array);
        addtocode(hi,code_array);
        break;

    case KtypeExpression:

        if(typeof(body.VtypeExpression->portName) == KTvoid)
        {
            op = typeExpressionOp;
                body.VtypeExpression->valuepos = currentpos;
                addtocode(op,code_array);

                /* calculate index into type reference array */
                index = add_type(SP,body.VtypeExpression->type);
            hi = (char) (index >> 8);
            lo = (char) index;
            addtocode(lo,code_array);
            addtocode(hi,code_array);
        }
        else    /* port name is present             */
        {
            op = portExpressionOp;
                body.VtypeExpression->valuepos = currentpos;
                addtocode(op,code_array);

                /* calculate index into type reference array */
            index = add_type(SP,body.VtypeExpression->type);
            hi = (char) (index >> 8);
            lo = (char) index;
            addtocode(lo,code_array);
            addtocode(hi,code_array);

                /* calculate reference into string reference array */
            index = add_String(SP,body.VtypeExpression
                            ->portName.VnameToken->name);
            hi = (char) (index >> 8);
            lo = (char) index;
            addtocode(lo,code_array);
            addtocode(hi,code_array);
        }

        break;

    case KdefnRef:

        /* generate code for all arguments of the application */
        foreachinSEQexpression(body.VdefnRef->arguments,Sarg,arg)
                gencode(arg,code_array);

        op = applicationOp;
        body.VdefnRef->valuepos = currentpos;
        addtocode(op,code_array);

        /* calculate index into definition reference array */
        index = add_definition(SP,body.VdefnRef->reference);
        hi = (char) (index >> 8);
        lo = (char) index;
        addtocode(lo,code_array);
        addtocode(hi,code_array);

        /* enter number of arguments into code array */
        op = len((pGenList)(body.VdefnRef->arguments));
        addtocode(op,code_array);

        break;

    case Kapplication:
       /* this is really type widening */
       /* ignore application and generate code for argument */

       retrievefirstSEQexpression(body.Vapplication->arguments,arg);
       gencode(arg,code_array);
       break;


    case KintegerToken:

       if(atoi(body.VintegerToken->integer) == 0)
       {
               op = ZeroOp;
               body.VintegerToken->valuepos = currentpos;
               addtocode(op,code_array);
       }
       else
       if(atoi(body.VintegerToken->integer) == 1)
       {
               op = OneOp;
               body.VintegerToken->valuepos = currentpos;
               addtocode(op,code_array);
       }
       else
       {
                   op = intToken;
                   body.VintegerToken->valuepos = currentpos;
                addtocode(op,code_array);

                /* calculate index into integer reference array */
                index = add_Integer(SP,atoi(body.VintegerToken->integer));
                hi = (char) (index >> 8);
                lo = (char) index;
                addtocode(lo,code_array);
                addtocode(hi,code_array);
        }
        break;

    case KrationalToken:

        op = ratToken;
        body.VrationalToken->valuepos = currentpos;
        addtocode(op,code_array);

        /* calculate index into rational reference array */
        index = add_Rational(SP,atof(body.VrationalToken->rational));
        hi = (char) (index >> 8);
        lo = (char) index;
        addtocode(lo,code_array);
        addtocode(hi,code_array);
        break;

    case KstringToken:

        op = strToken;
        body.VstringToken->valuepos = currentpos;
        addtocode(op,code_array);

        /* calculate index into string reference array */
        index = add_String(SP,body.VstringToken->string);
        hi = (char) (index >> 8);
        lo = (char) index;
        addtocode(lo,code_array);
        addtocode(hi,code_array);
        break;

    case Kroot:

        if(typeof(body.Vroot->portName) == KTvoid)
        {
                op = RootOp;
                body.Vroot->valuepos = currentpos;
                addtocode(op,code_array);
        }
        else    /* port name is present */
        {
            op = portRootOp;
                body.Vroot->valuepos = currentpos;
                addtocode(op,code_array);

                /* calculate index into string reference array */
            index = add_String(SP,body.Vroot->portName.VnameToken->name);
            hi = (char) (index >> 8);
            lo = (char) index;
            addtocode(lo,code_array);
            addtocode(hi,code_array);
        }

        break;

    case Kempty:

        op = emptyOp;
        body.Vempty->valuepos = currentpos;
        addtocode(op,code_array);
        break;

    case Ktrue:

        op = trueOp;
        body.Vtrue->valuepos = currentpos;
        addtocode(op,code_array);
        break;

    case Kfalse:

        op = falseOp;
        body.Vfalse->valuepos = currentpos;
        addtocode(op,code_array);
        break;

    case Kmembers:

        gencode(body.Vmembers->argument,code_array);
        op = membersOp;
        body.Vmembers->valuepos = currentpos;
        addtocode(op,code_array);
        break;

    case Khead:

        gencode(body.Vhead->argument,code_array);
        op = headOp;
        body.Vhead->valuepos = currentpos;
        addtocode(op,code_array);
        break;

    case Ktype:

        gencode(body.Vtype->argument,code_array);
        op = typeOp;
        body.Vtype->valuepos = currentpos;
        addtocode(op,code_array);
        break;

    case Ksize:

        gencode(body.Vsize->argument,code_array);
        if(typeof(body.Vsize->argument.IDLclassCommon->exp_type)
                == Ksingleton)
                tipe = typeof(body.Vsize->argument.IDLclassCommon
                                ->exp_type.Vsingleton->object_type);
        else
                tipe = typeof(body.Vsize->argument.IDLclassCommon->exp_type);

        if(tipe == Kstr)
            op = str_sizeOp;
        else
                op = setseq_sizeOp;

        body.Vsize->valuepos = currentpos;
        addtocode(op,code_array);
        break;

    case Ktail:

        gencode(body.Vtail->argument,code_array);
        op = tailOp;
        body.Vtail->valuepos = currentpos;
        addtocode(op,code_array);
        break;
```

```
case Kquantifier:

    level++;                        /* increment nesting level        */
    body.Vquantifier->nest_level = level;

    /* generate code for collection quantifier will iterate over */
    gencode(body.Vquantifier->set,code_array);

    if(typeof(body.Vquantifier->op) == Kforall)
            op = forallOp;
    else
            op = existsOp;

    addtocode(op,code_array);

    /* save start position for "return" statement that will be
       inserted at endquantifier operator */
    startpos = currentpos;

    /* generate code for body of quantifier */
    gencode(body.Vquantifier->body,code_array);

    body.Vquantifier->valuepos = currentpos;

    /* End operator will return to start position if there are
       more objects in the quantifier set                          */
    if(op == forallOp)
    {
            op = endForAll;
            addtocode(op,code_array);
            op = startpos + 1;
            addtocode(op,code_array);
    }
    else
    {
            op = endExists;
            addtocode(op,code_array);
            op = startpos + 1;
            addtocode(op,code_array);
    }

    level--;            /* decrement nesting level        */

    break;


case Kconditional:


    body.Vconditional->valuepos = currentpos;
    gencode(body.Vconditional->test,code_array);

    op = jfalse;
    addtocode(op,code_array);
    /* save current location to back insert later          */
    loc = currentpos;
    addtocode(loc,code_array);

    gencode(body.Vconditional->then,code_array);

    op = jump;
    addtocode(op,code_array);

    /* stack location of jump to backinsert exit position at
       end of conditional                                      */
    appendrearSEQInteger(loc_stack,currentpos);
```

```
            loc_count++;        /* increment count of how many positions
                                        to backinsert into later          */

            addtocode(currentpos,code_array);

            backinsert(currentpos,loc,code_array);

            /* handle OrIf pairs */
            foreachinSEQexpressionPair(body.Vconditional->orif,Sep,ep)
            {
                    gencode(ep->test,code_array);
                    op = jfalse;
                    addtocode(op,code_array);
                    loc = currentpos;
                    addtocode(loc,code_array);
                    gencode(ep->then,code_array);
                    op = jump;
                    addtocode(op,code_array);
                    appendrearSEQInteger(loc_stack,currentpos);
                    loc_count++;
                    addtocode(currentpos,code_array);
                    backinsert(currentpos,loc,code_array);
            }

            gencode(body.Vconditional->otherwise,code_array);

            /* backinsert exit location to all saved jump operator locations */
            for(i=1;i<=loc_count;i++)
                    backinsert(currentpos,pop(&loc_stack),code_array);

            break;


        case Kcontrol:

                    op = controlOp;
                    body.Vcontrol->valuepos = currentpos;
                    addtocode(op,code_array);

                    /* insert level of control operator */
                    op = body.Vcontrol->owner->nest_level;
                    addtocode(op,code_array);
                    break;

        case KformArg:

                    op = formArgOp;
                    body.VformArg->valuepos = currentpos;
                    addtocode(op,code_array);

                    /* insert position of formal arg in list of declared args */
                    op = position(body.VformArg->actual,in.IDLclassCommon->formals);
                    addtocode(op,code_array);
                break;


        }

}       /* end of generatecode */


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       addtocode               adds operator to code array              *
*                                                                          *
*          Last revised:          April 14,1986                                   *
*********************************************************************/

addtocode(entry,code)
int                   entry;                /* operator to add */
SEQInteger            *code;                /* code array         */
{
          appendrearSEQInteger(*code,entry);
          currentpos++;                     /* increment current position in code array */

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         backinsert insert location into a former location          *
*                                used to avoid backtracking                     *
*                                                                                *
*         Last revised:          April 14, 1986                                   *
*********************************************************************/


backinsert(l1,l2,code)
int                 l1;       /* location to insert */
int                 l2;       /* location where to insert */
SEQInteger          *code;    /* code array */
{

   int      i;
   pGenList            temp;

   temp = (pGenList)(*code);
   for(i=1;i<=l2;i++)
      temp = temp->next;

   temp->value = l1 + 1;
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         pop        pop a stack of integers                                     *
*                                                                                *
*         Last revised: April 14, 1986                                    *
*********************************************************************/

pop(stack)
SEQInteger            *stack;                /* stack to be popped */
{
   int                result;

   retrievelastSEQInteger(*stack,result);
   removelastSEQInteger(*stack);
   return(result);
}

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         len        len returns the length of a generic list             *
*                                                                                *
*         Last revised:          April 14, 1986                                   *
*********************************************************************/

len(list)
pGenList  list;        /* list to get length of */
{
          int                    result;
          pGenList  listptr;

          listptr = list;
          result = 0;
          while(listptr != NULL)
          {
             result++;
             listptr = listptr->next;
          }

          return(result);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         position    returns the position of a formal argument in a        *
*                             list of declared formal arguments                    *
*                                                                                  *
*         Last revised:        April 14, 1986                                      *
***************************************************************************/

position(form,declared_args)
formal              form;               /* formal to find position of */
SEQformal           declared_args; /* list of formal arguments */
{
          SEQformal           Sf;
          formal              f;
          int                 result;
          int                 i;

          i = 1;
          result = 0;

          foreachinSEQformal(declared_args,Sf,f)
          {
             if(f == form) result = i;
             i++;
          }

          return(result);

}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         add_Stringadd a string to string_refs array and return  *
*                             its position                                         *
*                                                                                  *
*         Last revised:        April 14, 1986                                      *
***************************************************************************/

add_String(SP,str)
SYMBOL SP;          /* structure or process containing string_refs array */
String    str;      /* string to add */
{
          int       result;

          if(typeof(SP) == Kstructure)
          {
                    appendrearSEQString(SP.Vstructure->string_refs,str);
                    result = len((pGenList)(SP.Vstructure->string_refs));
          }
          else
          {
                    appendrearSEQString(SP.Vprocess->string_refs,str);
                    result = len((pGenList)(SP.Vprocess->string_refs));
          }

          return(result);

}
```

```
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           add_Integer         add an integer to integer_refs array and return     *
*                               its position                                        *
*                                                                                     *
*           Last revised:       April 14, 1986                                        *
**********************************************************************/

add_Integer(SP,num)
SYMBOL SP;      /* structure or process containing integer_refs array */
int     num;    /* integer to add */
{

        int     result;

        if(typeof(SP) == Kstructure)
        {
                appendrearSEQInteger(SP.Vstructure->integer_refs,num);
                result = len((pGenList)(SP.Vstructure->integer_refs));
        }

        else
        {
                appendrearSEQInteger(SP.Vprocess->integer_refs,num);
                result = len((pGenList)(SP.Vprocess->integer_refs));
        }

        return(result);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           add_Rational        add a rational to rational_refs array and     *
*                               return its position                                 *
*                                                                                     *
*           Last revised:       April 14, 1986                                        *
**********************************************************************/

add_Rational(SP,num)
SYMBOL SP;      /* structure or process containing rational_refs array */
float   num;    /* rational to add */
{

        int     result;

        if(typeof(SP) == Kstructure)
        {
                appendrearSEQfloat(SP.Vstructure->rational_refs,num);
                result = len((pGenList)(SP.Vstructure->rational_refs));
        }
        else
        {
                appendrearSEQfloat(SP.Vprocess->rational_refs,num);
                result = len((pGenList)(SP.Vprocess->rational_refs));
        }

        return(result);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           add_type    add a type to type_refs array and               *
*                               return its position                                 *
*                                                                                     *
*           Last revised:       April 14, 1986                                        *
**********************************************************************/

add_type(SP,t)
SYMBOL SP;      /* structure or process containing type_refs array */
typeTree   t;   /* type to add */
```

```
{
        int       result;

        if(typeof(SP) == Kstructure)
        {
                appendrearSEQtypeTree(SP.Vstructure->type_refs,t);
                result = len((pGenList)(SP.Vstructure->type_refs));
        }
        else
        {
                appendrearSEQtypeTree(SP.Vprocess->type_refs,t);
                result = len((pGenList)(SP.Vprocess->type_refs));
        }

        return(result);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       add_definition      add a definition to define_refs array and    *
*                           return its position                          *
*                                                                         *
*       Last revised:       April 14, 1986                                *
*********************************************************************/

add_definition(SP,d)
SYMBOL SP;        /* structure or process containing define_refs array */
definition  d;    /* definition to add */
{

        int       result;

        if(typeof(SP) == Kstructure)
        {
                appendrearSEQDefOrInst(SP.Vstructure->define_refs,d);
                result = len((pGenList)(SP.Vstructure->define_refs));
        }
        else
        {
                appendrearSEQDefOrInst(SP.Vprocess->define_refs,d);
                result = len((pGenList)(SP.Vprocess->define_refs));
        }

        return(result);
}
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *                                                                      *
 *  Title:  Instructions for Assertion Checker                      *
 *  Filename:       ~kickenso/softlab/codegen/instructions.h        *
 *  Author:         Jerry Kickenson <kickenso@UNC>                      *
 *                  Department of Computer Science                      *
 *                  University of North Carolina                 *
 *                  Chapel Hill, NC  27514                              *
 *                                                                      *
 *  Copyright (C) The University of North Carolina, 1985            *
 *                                                                      *
 *  All rights reserved. No part of this software may be sold or   *
 *  distributed in any form or by any means without the prior written   *
 *  permission of the SoftLab Software Distribution Coordinator. *
 *                                                                      *
 *  Report problems to          softlab@unc (csnet) or                  *
 *                              softlab!unc@CSNET-RELAY (ARPAnet)    *
 *  Direct all inquiries to the SoftLab Software Distribution       *
 *          Coordinator, at the above addresses.                 *
 *                                                                      *
 *  Function: integer codes defined for checker instructions        *
 *                                                                      *
 *                                                                      *
 ***************************************************************** */

#define ZeroOp                  0
#define OneOp                   1
#define intToken        2
#define ratToken        3
#define strToken        4
#define trueOp                  5
#define falseOp                 6
#define emptyOp                 7
#define RootOp                  8
#define typeExpressionOp        9
#define portRootOp      10
#define portExpressionOp        11
#define unionOp                 12
#define intersectOp     13
#define subsetOp        14
#define propSubsetOp            15
#define plusOp                  16
#define minusOp                 17
#define timesOp                 18
#define divideOp        19
#define andOp                   20
#define orOp                    21
#define str_lessOp      22
#define str_lessEqOp 23
#define str_greaterOp 24
#define str_grtrEqOp 25
#define str_equalOp  26
#define str_notEqualOp          27
#define num_lessEqOp            28
#define num_lessOp  29
#define num_greaterOp           30
#define num_grtrEqOp            31
#define num_equalOp32
#define num_notEqualOp          33
#define bool_equalOp34
#define bool_notEqualOp         35
#define set_equalOp  36
#define set_notEqualOp          37
#define seq_equalOp 38
#define seq_notEqualOp          39
#define node_equalOp            40
#define node_notEqualOp         41
```

```
#define sameOp             42
#define inSetOp            43
#define inSeqOp            44
#define inCollectionOp     45
#define unaryMinusOp       46
#define notOp              47
#define jump               48
#define jtrue              49
#define jfalse             50
#define membersOp   51
#define headOp             52
#define tailOp             53
#define str_sizeOp  54
#define setseq_sizeOp55
#define typeOp             56
#define forallOp    57
#define existsOp    58
#define endForAll   59
#define endExists   60
#define controlOp   61
#define formArgOp   62
#define dotOp              63
#define applicationOp64
#define RETURN             65
#define ENDASSERTION       66
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *                                                                      *
 *  Title:  Interpreter for Assertion Checker                *
 *  Filename:       ~kickenso/softlab/check/check.c                   *
 *  Author:         Jerry Kickenson <kickenso@unc>                      *
 *                  Department of Computer Science                      *
 *                  University of North Carolina                  *
 *                  Chapel Hill, NC  27514                              *
 *                                                                      *
 *  Copyright (C) The University of North Carolina, 1985         *
 *                                                                      *
 *  All rights reserved. No part of this software may be sold or  *
 *  distributed in any form or by any means without the prior written      *
 *  permission of the SoftLab Software Distribution Coordinator. *
 *                                                                      *
 *  Report problems to          softlab@unc (csnet) or                  *
 *                              softlab!unc@CSNET-RELAY (ARPAnet)     *
 *  Direct all inquiries to the SoftLab Software Distribution         *
 *          Coordinator, at the above addresses.                      *
 *                                                                      *
 *  Function: main driver for interpreter for assertion checker        *
 *                                                                      *
 *                                                                      *
 ******************************************************************** */

static char rcsid = "$Header: check.c,v 1.8 86/04/09 08:37:43 kickenso Exp $"

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 * Revision Log:                                                        *
 *          $Log:     check.c,v $
 * Revision 1.8  86/04/09  08:37:43  kickenso
 * modifying for integer byte codes
 *
 * Revision 1.7  86/01/12  22:24:34  kickenso
 * rewrote code to handle Type function
 *
 * Revision 1.6  85/12/27  14:15:11  kickenso
 * added code to handle noncyclic definitions
 *
 * Revision 1.5  85/12/25  15:07:08  kickenso
 * Added code to handle all operators except formal args & applications
 *
 * Revision 1.4  85/12/24  17:07:31  kickenso
 * Added supplied functions
 * modified code to make all literals standard ...Desc as in idlData
 *
 * Revision 1.3  85/12/24  12:50:44  kickenso
 * Added more operators
 *
 * Revision 1.2  85/12/23  16:37:35  kickenso
 * added code to handle binary operators
 *
 * Revision 1.1  85/12/23  12:05:25  kickenso
 * Initial revision
 *                                                                  *
 *                                                                      *
 ***********************************************************************/


#include <stdio.h>
#include "Check.h"
#include "macros.h"
#include "../codegen/instructions.h"

typedef int             Operation;
typedef char                    BYTE;
typedef SEQInteger   SEQOperation;
```

```
#define stack(x,y)      stack_obj(x,y,code_pos-1);
#define MAXSTRUCS             10           /* maximum number of structures that can
                                              be read in for one process      */

runstackEntry          runtime_result[1000];  /* array which saves results of all
                                                   intermediate expressions.  Used by
                                                   routine that generates error log */
FILE      *errorfile;

main(argc,argv)
int         argc;
char        *argv[];
{

   runstackEntry      interpret();/* interpret may return a postfix code
                                             entry if a definition is interpreted */
   scope   code;               /* postfix code structure              */
   FILE    *in;                /* specific instances of structures  */
   struc allstrucs[MAXSTRUCS];         /* all structures read */

   struc in_data;      /* one instance of a read structure           */

   SETString         invs;                /* set of invariant structure names          */
   SEQSYMBOL        SSYMBOL;             /*  iterators ...                            */
   SYMBOL           ASymbol;
   SEQassertionStatement        Sas;
   assertionStatement           as;
   SEQOperation                 Spfc;
   Operation                                pfc;
   SETport                                  Spt;
   port                         pt;

   IDLVALUE                                 forms[50]; /* holds formal arguments */

   int                                      stindex;
   int                                      numstruc; /* number of instances            */
   FILE                         *fopen();
   char                         *malloc();
   boolean                      suppress;  /* true if detailed error
                                                      messages should be
                                                      suppressed                */

suppress = FALSE;

   /* read in postfix code         */
   code = Code(stdin);

/* read in all structure instances */

   numstruc = -1;
   for(stindex=1;stindex<argc;stindex++)
   {
          if(streq(argv[stindex],"-w"))
             suppress = TRUE;
          else
          {
             in_data.portname = malloc(strlen(argv[stindex])+1);
             strcpy(in_data.portname,argv[stindex]);
             if(streq(argv[++stindex],"-"))
                    in_data.st = data_in(stdin);
             else
             {
                if((in = fopen(argv[stindex],"r")) == NULL)
                       fprintf(stderr,"File %s cannot be opened for reading.0,
                                        argv[stindex]);
                in_data.st = data_in(in);
             }
```

```
            allstrucs[++numstruc] = in_data;
        }
}


invs = NULL;
/* collect all invariant structure names and add structure names
   to structure instances */

foreachinSEQSYMBOL(code->symbols,SSYMBOL,ASymbol)
{
        if(typeof(ASymbol) == Kprocess)
        {
            addSETString(invs,ASymbol.Vprocess->invariant->name);
            for(stindex=0;stindex<=numstruc;stindex++)
               foreachinSETport(ASymbol.Vprocess->ports,Spt,pt)
                  if(streq(pt.IDLclassCommon->name,
                              allstrucs[stindex].portname))
                  {
                        allstrucs[stindex].stname =
                                malloc(strlen(pt.IDLclassCommon->data->name)+1);
                        strcpy(allstrucs[stindex].stname,
                                pt.IDLclassCommon->data->name);
                      }
        }
}


/* interpret all assertions                    */
foreachinSEQSYMBOL(code->symbols,SSYMBOL,ASymbol)
{
        /* first handle assertions within structures, excepting invariants */
        if(typeof(ASymbol) == Kstructure)
        {
           if(!inSETString(invs,ASymbol.Vstructure->name))
           {
                  foreachinSEQassertionStatement(ASymbol.Vstructure->assertions,
                                                          Sas,as)
                     if(typeof(as) == Kassertion)
                     {
                            stindex = 0;
                            while(stindex <= numstruc)
                            {
                               if(streq(ASymbol.Vstructure->name,
                                                          allstrucs[stindex].stname))
                                      check(as.Vassertion->postfixBody,
                                             as.Vassertion->name,ASymbol,
                                             as.IDLclassCommon->sourceposition,
                                             as.Vassertion->body);
                               stindex++;
                            }
                     }
           }
        /* now handle assertions within processes   */
        }
        else if(typeof(ASymbol) == Kprocess)
        {
           foreachinSEQassertionStatement(ASymbol.Vprocess->assertions,
                                                          Sas,as)
                  if(typeof(as) == Kassertion)
                  {
                     check(as.Vassertion->postfixBody,as.Vassertion->name,
                                    ASymbol,as.IDLclassCommon->sourceposition,
                                    as.Vassertion->body);
                  }
  }
```

```
    }

}         /* end of main */



/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*         interpret   interpret assertion code in given code array.*
*                             Return result.  Interpretation of an assertion*
*                             code array always results in a boolean value.        *
*                             Interpretation of a definition instance code  *
*                             array may result in any runstack entry type.*
*                             If an assertion returns false, an error      *
*                             log describing the failed assertion is               *
*                             generated.  The error log generation involves       *
*                             a walk of the expression tree.                       *
*                                                                                  *
*         Last revised:       June 7, 1986                                         *
*******************************************************************/


runstackEntry        interpret(body,name,SP,spos,forms,allstrucs,
                               numstruc,stindex,tree,suppress)
SEQInteger           body;     /* code array to be interpreted */
nameOrVoid           name;     /* assertion name, if any          */
SYMBOL               SP;       /* structure/process assertion is in */
int                  spos;     /* sourceposition of assertion */
IDLVALUE             forms[]; /* list of definition arguments - used
                                 when interpreting definition instance body */
struc                allstrucs[]; /* array of structure instances */
int                  numstruc;  /* number of structures in allstrucs */
int                  stindex;   /* index of structure in allstrucs */
expression tree;     /* expression tree for assertion - used in
                               generating error log */
boolean              suppress; /* TRUE if error log should be suppressed */
{

    runstackEntry    pop();                        /* pop the runtime stack */
    runstackEntry    get_object();                 /* returns object in a
                                                            singleton collection */
    SEQrunstackEntry runstack = NULL;   /* run time stack       */
    int                      code_pos = 1;                 /* code array position*/
    Operation                entry;                        /* current entry in code */
    IDLVALUE                 controlArray[50];  /* holds values controls
                                                            will take on                    */
    SEQInteger               controlReturn = NULL;         /* return points in control
                                                            array after applications */
    int                      offset  = 0;                  /* offset in controlArray */
    int                      numcontrol = 0;    /* number of objects in control
                                                  array                    */
    int                      level;             /* level of control reference      */
    int                      nestlevel = 0;     /* nesting level of quantifier     */
    int                      formoffset = 0;               /* offset in formalArray */
    int                      numargs = 0;                  /* number of arguments in
                                                            formalArray                     */
    int                      pos;                          /* position of formal in
                                                            formal array                    */
    IDLVALUE                 formalArray[50];   /* holds values formals
                                                            will take on                    */
    SEQIDLVALUE              args = NULL;                  /* application arguments  */


    runstackEntry    op1;                          /* operand */
    runstackEntry    op2;                          /* operand */
    runstackEntry    result;                       /* result */
```

```
SEQcollect          quantsets = NULL;   /* stores quantifier sets */
collect     qs;

SEQIDLVALUE         Sobj;               /*  iterators ...               */
IDLVALUE            obj;
IDLVALUE            obj2;
int                 i;
SEQattrDesc         Sad;
attrDesc            ad;
SETString           Ss;
String      s;

int                 hi,lo;              /* high,low bytes of reference index */
int                 index;              /* index into reference array    */

pGenList            listptr1;  /* temporary to hold sets,sequences */
pGenList            listptr2;  /*                 "                   */
int                 eq,eeq;             /* temporary to hold current value of
                                           T/F for equality/inequality      */
IDLVALUE            hed;                /* holds value of head of a sequence */
SETString           types;              /* holds all types reachable from a
                                           another type, which may be a class */

IDLVALUE            arg;                /* a single argument            */
runstackEntry   argrun;        /* argument when on runstack   */
idlInstance         instance;  /* definition instance         */
idlInstance         findinstance();     /* finds instance of a definition
                                           which applies to arguments      */
IDLVALUE            runstack_to_value();
                                        /* translates runstack entry to
                                           equivalent IDLVALUE                    */
runstackEntry   value_to_runstack();
                                        /* translates IDLVALUE to equivalent
                                           runstack entry                  */

nodeDesc            strc;               /* structure instance           */

SEQString           string_refs;        /* string references   */
SEQInteger          integer_refs;       /* integer references  */
SEQfloat            rational_refs;      /* rational references */
SEQtypeTree         type_refs; /* type references      */
SEQDefOrInst    define_refs;       /* definition references */
int                 int_value; /* integer value       */
float       rat_value; /* rational value      */
String      str_value; /* string value                 */
typeTree            type_value;         /* type value                   */
DefOrInst           def_value; /* definition value     */

SEQInteger          Sn;
int                 n;

/* initialize control and formal return arrays */
appendfrontSEQInteger(controlReturn,0);

/* assign correct reference arrays */
if(typeof(SP) == Kstructure)
{
        string_refs = SP.Vstructure->string_refs;
        integer_refs = SP.Vstructure->integer_refs;
        rational_refs = SP.Vstructure->rational_refs;
        type_refs = SP.Vstructure->type_refs;
        define_refs = SP.Vstructure->define_refs;
}
else
{
        string_refs = SP.Vprocess->string_refs;
        integer_refs = SP.Vprocess->integer_refs;
        rational_refs = SP.Vprocess->rational_refs;
        type_refs = SP.Vprocess->type_refs;
        define_refs = SP.Vprocess->define_refs;
}


/* get first code operator */
ithinSEQInteger(body,code_pos++,entry);

/* ENDASSERTION ends an assertion interpretation.
   RETURN ends a definition instance interpretation. */

while(entry != ENDASSERTION && entry != RETURN)
{
        switch (entry)  {

            case ZeroOp:
                    result.VintegerDesc = NintegerDesc;
                    result.VintegerDesc->value = 0;
                    stack(result,&runstack);
                    break;
            case OneOp:
                    result.VintegerDesc = NintegerDesc;
                    result.VintegerDesc->value = 1;
                    stack(result,&runstack);
                    break;

            /* binary operators */
            case unionOp:

                    result.Vcollect = Ncollect;

                    op2 = pop(&runstack);
                    op1 = pop(&runstack);
                    foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
                        appendrearSEQIDLVALUE(result.Vcollect->objects,obj);
                    foreachinSEQIDLVALUE(op2.Vcollect->objects,Sobj,obj)
                        if(!inSEQIDLVALUE(result.Vcollect->objects,obj))
                                appendrearSEQIDLVALUE(result.Vcollect->objects,obj);

                    stack(result,&runstack);
                    break;

            case intersectOp:

                    result.Vcollect = Ncollect;

                    op2 = pop(&runstack);
                    op1 = pop(&runstack);
                    foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
                        if(inSEQIDLVALUE(op2.Vcollect->objects,obj))
                                appendrearSEQIDLVALUE(result.Vcollect->objects,obj);

                    stack(result,&runstack);
                    break;

            case plusOp:

                    op2 = pop(&runstack);
                    op1 = pop(&runstack);

                    /* singleton collections */
                    if(typeof(op1) == Kcollect)
                        op1 = get_object(op1);
                    if(typeof(op2) == Kcollect)
                        op2 = get_object(op2);
        if(typeof(op1) == KrationalDesc)
        {
            result.VrationalDesc = NrationalDesc;
            if(typeof(op2) == KrationalDesc)
                    result.VrationalDesc->value = op1.VrationalDesc->value +
                                                   op2.VrationalDesc->value;
            else
                    result.VrationalDesc->value = op1.VrationalDesc->value +
                                                               op2.VintegerDesc->value;
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
            {
                    result.VrationalDesc = NrationalDesc;
                    result.VrationalDesc->value = op1.VintegerDesc->value +
                                                   op2.VrationalDesc->value;
            }
            else
            {
                    result.VintegerDesc = NintegerDesc;
                    result.VintegerDesc->value = op1.VintegerDesc->value +
                                                               op2.VintegerDesc->value;
            }
        }

        stack(result,&runstack);
        break;

case minusOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) == KrationalDesc)
        {
            result.VrationalDesc = NrationalDesc;
            if(typeof(op2) == KrationalDesc)
                    result.VrationalDesc->value = op1.VrationalDesc->value -
                                                   op2.VrationalDesc->value;
            else
                    result.VrationalDesc->value = op1.VrationalDesc->value -
                                                               op2.VintegerDesc->value;
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
            {
                    result.VrationalDesc = NrationalDesc;
                    result.VrationalDesc->value = op1.VintegerDesc->value -
                                                   op2.VrationalDesc->value;
            }
            else
            {
                    result.VintegerDesc = NintegerDesc;
                    result.VintegerDesc->value = op1.VintegerDesc->value -
                                                               op2.VintegerDesc->value;
            }
        }

        stack(result,&runstack);
        break;

case timesOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) == KrationalDesc)
        {
            result.VrationalDesc = NrationalDesc;
            if(typeof(op2) == KrationalDesc)
                    result.VrationalDesc->value = op1.VrationalDesc->value *
                                                  op2.VrationalDesc->value;
            else
                    result.VrationalDesc->value = op1.VrationalDesc->value *
                                                            op2.VintegerDesc->value;
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
            {
                    result.VrationalDesc = NrationalDesc;
                    result.VrationalDesc->value = op1.VintegerDesc->value *
                                                  op2.VrationalDesc->value;
            }
            else
            {
                    result.VintegerDesc = NintegerDesc;
                    result.VintegerDesc->value = op1.VintegerDesc->value *
                                                            op2.VintegerDesc->value;
            }
        }

        stack(result,&runstack);
        break;

case divideOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        result.VrationalDesc = NrationalDesc;
        if(typeof(op1) == KrationalDesc)
        {
            if(typeof(op2) == KrationalDesc)
                    result.VrationalDesc->value = op1.VrationalDesc->value /
                                                  op2.VrationalDesc->value;
            else
                    result.VrationalDesc->value = op1.VrationalDesc->value /
                                                            op2.VintegerDesc->value;
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
                    result.VrationalDesc->value = op1.VintegerDesc->value /
```

```
                                        op2.VrationalDesc->value;
            else
                    result.VrationalDesc->value = op1.VintegerDesc->value /
                                                        op2.VintegerDesc->value;
        }

        stack(result,&runstack);
        break;


    case andOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) == KTVALUE && typeof(op2) == KTVALUE)
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

    case orOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) == KTVALUE || typeof(op2) == KTVALUE)
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

    case str_lessOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(strcmp(op1.VstringDesc->value,op2.VstringDesc->value) < 0)
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;
```

```
case num_lessOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) == KrationalDesc)
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VrationalDesc->value < op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VrationalDesc->value < op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VintegerDesc->value < op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VintegerDesc->value < op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }

        stack(result,&runstack);
        break;

case str_lessEqOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(strcmp(op1.VstringDesc->value,op2.VstringDesc->value) <= 0)
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;
```

```
case num_lessEqOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) == KrationalDesc)
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VrationalDesc->value <= op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VrationalDesc->value <= op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VintegerDesc->value <= op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VintegerDesc->value <= op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }

        stack(result,&runstack);
        break;

case str_greaterOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(strcmp(op1.VstringDesc->value,op2.VstringDesc->value) > 0)
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;
```

```
case num_greaterOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
           op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
           op2 = get_object(op2);

        if(typeof(op1) == KrationalDesc)
        {
           if(typeof(op2) == KrationalDesc)
           {
                   if(op1.VrationalDesc->value > op2.VrationalDesc->value)
                      result.VTVALUE = NTVALUE;
                   else
                      result.VFVALUE = NFVALUE;
           }
           else
           {
                   if(op1.VrationalDesc->value > op2.VintegerDesc->value)
                      result.VTVALUE = NTVALUE;
                   else
                      result.VFVALUE = NFVALUE;
           }
        }
        else
        {
           if(typeof(op2) == KrationalDesc)
           {
                   if(op1.VintegerDesc->value > op2.VrationalDesc->value)
                      result.VTVALUE = NTVALUE;
                   else
                      result.VFVALUE = NFVALUE;
           }
           else
           {
                   if(op1.VintegerDesc->value > op2.VintegerDesc->value)
                      result.VTVALUE = NTVALUE;
                   else
                      result.VFVALUE = NFVALUE;
           }
        }

        stack(result,&runstack);
        break;

case str_grtrEqOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
           op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
           op2 = get_object(op2);

        if(strcmp(op1.VstringDesc->value,op2.VstringDesc->value) >= 0)
           result.VTVALUE = NTVALUE;
        else
           result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;
```

```
case num_grtrEqOp:
        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) == KrationalDesc)
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VrationalDesc->value >= op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VrationalDesc->value >= op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VintegerDesc->value >= op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VintegerDesc->value >= op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }

        stack(result,&runstack);
        break;

case bool_equalOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) == typeof(op2))
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;
```

```
case num_equalOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);



        if(typeof(op1) == KrationalDesc)
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VrationalDesc->value == op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VrationalDesc->value == op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VintegerDesc->value == op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VintegerDesc->value == op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }

        stack(result,&runstack);
        break;

case str_equalOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(streq(op1.VstringDesc->value,op2.VstringDesc->value))
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;
            stack(result,&runstack);
            break;

    case set_equalOp:

            op2 = pop(&runstack);
            op1 = pop(&runstack);

            /* singleton collections */
            if(typeof(op1) == Kcollect)
                op1 = get_object(op1);
            if(typeof(op2) == Kcollect)
                op2 = get_object(op2);

            eq = FALSE;           /* is one object in the other set? */
            eeq = TRUE;           /* do both sets contain the same objects? */
            listptr1 = (pGenList)(op1.VsetDesc->value);
            listptr2 = (pGenList)(op2.VsetDesc->value);

            /* is every object in 1st set also in 2nd set? */
            while(listptr1 != NULL && eeq == TRUE)
            {
                while(listptr2 != NULL)
                {
                    if(listptr1->value == listptr2->value)
                            eq = TRUE;
                    listptr2 = listptr2->next;
                }
                if(eq == FALSE)
                        eeq = FALSE;
                eq = FALSE;
                listptr1 = listptr1->next;
            }

            if(eeq == FALSE)
                result.VFVALUE = NFVALUE;
            else
            {
                eq = FALSE;
                listptr1 = (pGenList)(op1.VsetDesc->value);
                listptr2 = (pGenList)(op2.VsetDesc->value);

                /* is every object in 2nd set also in 1st set? */
                while(listptr2 != NULL && eeq == TRUE)
                {
                    while(listptr1 != NULL)
                    {
                        if(listptr2->value == listptr2->value)
                                eq = TRUE;
                        listptr1 = listptr1->next;
                    }
                    if(eq == FALSE)
                            eeq = FALSE;
                    eq = FALSE;
                    listptr2 = listptr2->next;
                }

                if(eeq == FALSE)
                        result.VFVALUE = NFVALUE;
                else
                        result.VTVALUE = NTVALUE;
            }

            stack(result,&runstack);
            break;

    case seq_equalOp:
        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        listptr1 == (pGenList)(op1.VsequenceDesc->value);
        listptr2 == (pGenList)(op2.VsequenceDesc->value);

        eq = TRUE;

        /* if lengths are unequal, sequences are not equal */
        if(len(listptr1) != len(listptr2))
            result.VFVALUE = NFVALUE;
        else        /* are objects in corresponding positions in each
                       sequence the same? */
        {
            while(listptr1 != NULL && listptr2 != NULL)
            {
                     if(listptr1->value != listptr2->value)
                         eq = FALSE;
                     listptr1 = listptr1->next;
                     listptr2 = listptr2->next;
            }

            if(eq == FALSE)
                     result.VFVALUE = NFVALUE;
            else
                     result.VTVALUE = NTVALUE;
        }

        stack(result,&runstack);
        break;

case node_equalOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        if(streq(op1.VnodeDesc->label,op2.VnodeDesc->label))
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

case bool_notEqualOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(typeof(op1) != typeof(op2))
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

  case num_notEqualOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);


        if(typeof(op1) == KrationalDesc)
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VrationalDesc->value != op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VrationalDesc->value != op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }
        else
        {
            if(typeof(op2) == KrationalDesc)
            {
                    if(op1.VintegerDesc->value != op2.VrationalDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
            else
            {
                    if(op1.VintegerDesc->value != op2.VintegerDesc->value)
                        result.VTVALUE = NTVALUE;
                    else
                        result.VFVALUE = NFVALUE;
            }
        }


        stack(result,&runstack);
        break;

  case str_notEqualOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        if(!streq(op1.VstringDesc->value,op2.VstringDesc->value))
            result.VTVALUE = NTVALUE;
        else
```

```
          result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

case set_notEqualOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        eq = FALSE;         /* is one object in the other set? */
        eeq = TRUE;         /* do both sets contain the same objects? */
        listptr1 = (pGenList)(op1.VsetDesc->value);
        listptr2 = (pGenList)(op2.VsetDesc->value);

        /* is every object in 1st set also in 2nd set? */
        while(listptr1 != NULL && eeq == FALSE)
        {
            while(listptr2 != NULL)
            {
                if(listptr1->value == listptr2->value)
                        eq = TRUE;
                listptr2 = listptr2->next;
            }
            if(eq == FALSE)
                    eeq = FALSE;
            eq = FALSE;
            listptr1 = listptr1->next;
        }

        if(eeq == FALSE)
            result.VTVALUE = NTVALUE;
        else
        {
            eq = FALSE;
            listptr1 = (pGenList)(op1.VsetDesc->value);
            listptr2 = (pGenList)(op2.VsetDesc->value);

            /* is every object in 2nd set also in 1st set? */
            while(listptr2 != NULL && eeq == FALSE)
            {
                while(listptr1 != NULL)
                {
                    if(listptr2->value == listptr2->value)
                            eq = TRUE;
                    listptr1 = listptr1->next;
                }
                if(eq == FALSE)
                        eeq = FALSE;
                eq = FALSE;
                listptr2 = listptr2->next;
            }

            if(eeq == FALSE)
                    result.VTVALUE = NTVALUE;
            else
                    result.VFVALUE = NFVALUE;
        }

        stack(result,&runstack);
        break;
```

```
case seq_notEqualOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collection */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        listptr1 == (pGenList)(op1.VsequenceDesc->value);
        listptr2 == (pGenList)(op2.VsequenceDesc->value);

        eq = TRUE;

        if(len(listptr1) != len(listptr2))
            result.VTVALUE = NTVALUE;
        else
        {
            while(listptr1 != NULL && listptr2 != NULL)
            {
                    if(listptr1->value != listptr2->value)
                        eq = FALSE;
                    listptr1 = listptr1->next;
                    listptr2 = listptr2->next;
            }

            if(eq == FALSE)
                    result.VTVALUE = NTVALUE;
            else
                    result.VFVALUE = NFVALUE;
        }

        stack(result,&runstack);
        break;

case node_notEqualOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        if(!streq(op1.VnodeDesc->label,op2.VnodeDesc->label))
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

case sameOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        eq = TRUE;

        /* check for Type operand */
        if(typeof(op1) != Kcollect || typeof(op2) != Kcollect)
        {
            if(typeof(op1) == Kcollect)
                    retrievefirstSEQIDLVALUE(op1.Vcollect->objects,obj);
            else
                    obj = runstack_to_value(op1);
            if(typeof(op2) == Kcollect)
                    retrievefirstSEQIDLVALUE(op2.Vcollect->objects,obj2);
            else
```

```
                    obj2 = runstack_to_value(op2);

                if(typeof(obj) != typeof(obj2))
                    eq = FALSE;
                else
                {
                    if(typeof(obj) == KnodeDesc)
                        if(!streq(obj.VnodeDesc->name,obj2.VnodeDesc->name))
                            eq = FALSE;

                    if(typeof(obj) == KsetDesc)
                        if(!set_subset(obj.VsetDesc->value,
                                        obj2.VsetDesc->value)
                            || !set_subset(obj2.VsetDesc->value,
                                            obj.VsetDesc->value))
                            eq = FALSE;

                    if(typeof(obj) == KsequenceDesc)
                        if(!seq_subset(obj.VsequenceDesc->value,
                                        obj2.VsequenceDesc->value)
                            || !seq_subset(obj2.VsequenceDesc->value,
                                            obj.VsequenceDesc->value))
                            eq = FALSE;
                }
            }
            else /* both operands are collections */
            {
            /* if lengths unequal, collections are not same */
            if(len((pGenList)(op1.Vcollect->objects)) !=
              len((pGenList)(op2.Vcollect->objects)))
            eq = FALSE;
            else        /* check that each is included in the other */
            {
                foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
                    if(!inSEQIDLVALUE(op2.Vcollect->objects,obj))
                        eq = FALSE;
                if(eq == TRUE)
                {
                    foreachinSEQIDLVALUE(op2.Vcollect->objects,Sobj,obj)
                        if(!inSEQIDLVALUE(op2.Vcollect->objects,obj))
                            eq = FALSE;
                }
            }
            }

            if(eq == TRUE)
                result.VTVALUE = NTVALUE;
            else
                result.VFVALUE = NFVALUE;

            stack(result,&runstack);
            break;

        case inSetOp:

            op2 = pop(&runstack);
            op1 = pop(&runstack);

            /* singleton collections */
            if(typeof(op1) == Kcollect)
                op1 = get_object(op1);
            if(typeof(op2) == Kcollect)
                op2 = get_object(op2);

            eq == FALSE;

            if(typeof(op1) == KintegerDesc)
            if(inSETIDLVALUE(op1.VintegerDesc,op2.VsetDesc->value))
                    eq = TRUE;
        if(typeof(op1) == KrationalDesc)
            if(inSETIDLVALUE(op1.VrationalDesc,op2.VsetDesc->value))
                    eq = TRUE;
        if(typeof(op1) == KstringDesc)
            if(inSETIDLVALUE(op1.VstringDesc,op2.VsetDesc->value))
                    eq = TRUE;
        if(typeof(op1) == KbooleanDesc)
            if(inSETIDLVALUE(op1.VbooleanDesc,op2.VsetDesc->value))
                    eq = TRUE;
        if(typeof(op1) == KnodeDesc)
            if(inSETIDLVALUE(op1.VnodeDesc,op2.VsetDesc->value))
                    eq = TRUE;

        if(eq == TRUE)
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

case inSeqOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        if(typeof(op2) == Kcollect)
            op2 = get_object(op2);

        eq == FALSE;

        if(typeof(op1) == KintegerDesc)
            if(inSEQIDLVALUE(op1.VintegerDesc,op2.VsequenceDesc->value))
                    eq = TRUE;
        if(typeof(op1) == KrationalDesc)
            if(inSEQIDLVALUE(op1.VrationalDesc,op2.VsequenceDesc
                                                            ->value))
                    eq = TRUE;
        if(typeof(op1) == KstringDesc)
            if(inSEQIDLVALUE(op1.VstringDesc,op2.VsequenceDesc->value))
                    eq = TRUE;
        if(typeof(op1) == KbooleanDesc)
            if(inSEQIDLVALUE(op1.VbooleanDesc,op2.VsequenceDesc->value))
                    eq = TRUE;
        if(typeof(op1) == KnodeDesc)
            if(inSEQIDLVALUE(op1.VnodeDesc,op2.VsequenceDesc->value))
                    eq = TRUE;

        if(eq == TRUE)
            result.VTVALUE = NTVALUE;
        else
            result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

case inCollectionOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        /* singleton collections */
/* Unary operators*/
case unaryMinusOp:

        op1 = pop(&runstack);

        /* singleton collection */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);

        if(typeof(op1) == KrationalDesc)
        {
            result.VrationalDesc = NrationalDesc;
            result.VrationalDesc->value = -1*(op1.VrationalDesc->value);
        }
        else
        {
            result.VintegerDesc = NintegerDesc;
            result.VintegerDesc->value = -1 * (op1.VintegerDesc->value);
        }

        stack(result,&runstack);
        break;

case notOp:

        op1 = pop(&runstack);

        /* singleton collection */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);

        if(typeof(op1) == KrationalDesc)
        if(typeof(op1) == KTVALUE)
            result.VFVALUE = NFVALUE;
        else
            result.VTVALUE = NTVALUE;

        stack(result,&runstack);
        break;

/* Jump operations            */
case jump:

        ithinSEQInteger(body,code_pos++,entry);
        code_pos = entry;
        break;

case jtrue:

        ithinSEQInteger(body,code_pos++,entry);
        op1 = pop(&runstack);
        if(typeof(op1) == KTVALUE)
            code_pos = entry;
        break;

case jfalse:

        ithinSEQInteger(body,code_pos++,entry);
        op1 = pop(&runstack);
        if(typeof(op1) == KFVALUE)
            code_pos = entry;
        break;


/* Literal entries  */
case intToken:
        if(typeof(op1) == Kcollect)
           op1 = get_object(op1);

        eq == FALSE;

        if(typeof(op1) == KintegerDesc)
           if(inSEQIDLVALUE(op1.VintegerDesc,op2.Vcollect->objects))
                  eq = TRUE;
        if(typeof(op1) == KrationalDesc)
           if(inSEQIDLVALUE(op1.VrationalDesc,
                                          op2.Vcollect->objects))
                  eq = TRUE;
        if(typeof(op1) == KstringDesc)
           if(inSEQIDLVALUE(op1.VstringDesc,op2.Vcollect->objects))
                  eq = TRUE;
        if(typeof(op1) == KbooleanDesc)
           if(inSEQIDLVALUE(op1.VbooleanDesc,op2.Vcollect->objects))
                  eq = TRUE;
        if(typeof(op1) == KnodeDesc)
           if(inSEQIDLVALUE(op1.VnodeDesc,op2.Vcollect->objects))
                  eq = TRUE;

        if(eq == TRUE)
           result.VTVALUE = NTVALUE;
        else
           result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

case subsetOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        eq = TRUE;
        foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
           if(!inSEQIDLVALUE(op2.Vcollect->objects,obj))
                  eq = FALSE;

        if(eq == TRUE && len((pGenList)(op1.Vcollect->objects))
                           <= len((pGenList)(op2.Vcollect->objects)))
           result.VTVALUE = NTVALUE;
        else
           result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;

case propSubsetOp:

        op2 = pop(&runstack);
        op1 = pop(&runstack);

        eq = TRUE;
        foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
           if(!inSEQIDLVALUE(op2.Vcollect->objects,obj))
                  eq = FALSE;

        if(eq == TRUE && len((pGenList)(op1.Vcollect->objects))
                           < len((pGenList)(op2.Vcollect->objects)))
           result.VTVALUE = NTVALUE;
        else
           result.VFVALUE = NFVALUE;

        stack(result,&runstack);
        break;
        /* get index into integer array */
        ithinSEQInteger(body,code_pos++,lo);
        ithinSEQInteger(body,code_pos,hi);
        index = (hi << 8) | lo;

        /* retrieve value */
        ithinSEQInteger(integer_refs,index,int_value);
        result.VintegerDesc = NintegerDesc;
        result.VintegerDesc->value = int_value;

        code_pos--; /* decrement for accurate runtime_result entry */
        stack(result,&runstack);
        code_pos = code_pos + 2;        /* adjust to next operator */
        break;

case ratToken:

        /* get index into rational array */
        ithinSEQInteger(body,code_pos++,lo);
        ithinSEQInteger(body,code_pos,hi);
        index = (hi << 8) | lo;

        /* retrieve value */
        ithinSEQfloat(rational_refs,index,rat_value);
        result.VrationalDesc = NrationalDesc;
        result.VrationalDesc->value = rat_value;

        code_pos--; /* decrement for accurate runtime_result entry */
        stack(result,&runstack);
        code_pos = code_pos+2;          /* adjust to next operator */
        break;

case strToken:

        /* get index into string array */
        ithinSEQInteger(body,code_pos++,lo);
        ithinSEQInteger(body,code_pos,hi);
        index = (hi << 8) | lo;

        /* retrieve value */
        ithinSEQString(string_refs,index,str_value);
        result.VstringDesc = NstringDesc;
        result.VstringDesc->value = str_value;

        code_pos--; /* decrement for accurate runtime_result entry */
        stack(result,&runstack);
        code_pos = code_pos+2;          /* adjust to next operator */
        break;


case trueOp:

        result.VTVALUE = NTVALUE;
        stack(result,&runstack);
        break;

case falseOp:

        result.VFVALUE = NFVALUE;
        stack(result,&runstack);
        break;

case emptyOp:

        result.Vcollect = Ncollect;
        result.Vcollect->objects = NULL;
        stack(result,&runstack);
```

```
          break;

 case RootOp:

          result.Vcollect = Ncollect;

          get_structure(&strc,allstrucs,stindex);
          appendrearSEQIDLVALUE(result.Vcollect->objects,strc);
          stack(result,&runstack);

          break;

 case portRootOp:

          result.Vcollect = Ncollect;

          /* get index into string array */
          ithinSEQInteger(body,code_pos++,lo);
          ithinSEQInteger(body,code_pos,hi);
          index = (hi << 8) | lo;

          /* retrieve value */
          ithinSEQString(string_refs,index,str_value);

          get_structure_port(&strc,str_value,allstrucs,numstruc);
          appendrearSEQIDLVALUE(result.Vcollect->objects,strc);

          code_pos--;
          stack(result,&runstack);
          code_pos = code_pos + 2;
          break;


 /* Supplied Functions         */
 case membersOp:

          op1 = pop(&runstack);
          if(typeof(op1) == Kcollect)
              op1 = get_object(op1);

          result.Vcollect = Ncollect;
          if(typeof(op1) == KsetDesc)
              foreachinSETIDLVALUE(op1.VsetDesc->value,Sobj,obj)
                  appendrearSEQIDLVALUE(result.Vcollect->objects,obj);
          else /* sequence argument */
              foreachinSEQIDLVALUE(op1.VsequenceDesc->value,Sobj,obj)
                  appendrearSEQIDLVALUE(result.Vcollect->objects,obj);


          stack(result,&runstack);
          break;

 case headOp:

          result.Vcollect = Ncollect;
          op1 = pop(&runstack);

          /* singleton collect */
          if(typeof(op1) == Kcollect)
              op1 = get_object(op1);
          retrievefirstSEQIDLVALUE(op1.VsequenceDesc->value,hed);
          appendrearSEQIDLVALUE(result.Vcollect->objects,hed);

          stack(result,&runstack);
          break;

 case tailOp:
        op1 = pop(&runstack);

        /* singleton collect */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        result.VsequenceDesc = NsequenceDesc;
        result.VsequenceDesc->value = NULL;
        for(i=2;i<=len((pGenList)(op1.VsequenceDesc->value));i++)
        {
            ithinSEQIDLVALUE(op1.VsequenceDesc->value,i,obj);
            appendrearSEQIDLVALUE(result.VsequenceDesc->value,obj);
        }

        stack(result,&runstack);
        break;

case str_sizeOp:

        op1 = pop(&runstack);

        /* singleton collect */
        if(typeof(op1) == Kcollect)
            op1 = get_object(op1);
        result.VintegerDesc = NintegerDesc;
        result.VintegerDesc->value = strlen(op1.VstringDesc->value);

        stack(result,&runstack);
        break;

case setseq_sizeOp:

        op1 = pop(&runstack);

        /* singleton collect with set or sequence object */
        if(typeof(op1) == Kcollect)
            if(len((pGenList)(op1.Vcollect->objects)) == 1)
            {
                    retrievefirstSEQIDLVALUE(op1.Vcollect->objects,obj);
                    if(typeof(obj) == KsetDesc ||
                       typeof(obj) == KsequenceDesc)
                        op1 = get_object(op1);
            }

        result.VintegerDesc = NintegerDesc;

        if(typeof(op1) == KsetDesc)
            result.VintegerDesc->value =
                            len((pGenList)(op1.VsetDesc->value));

        if(typeof(op1) == KsequenceDesc)
            result.VintegerDesc->value =
                            len((pGenList)(op1.VsequenceDesc->value));

        if(typeof(op1) == Kcollect)
            result.VintegerDesc->value =
                            len((pGenList)(op1.Vcollect->objects));

        stack(result,&runstack);
        break;

case typeOp:
        op1 = pop(&runstack);

        /* if Type is in process, then typename is meant since
         a collection doesn't make sense */
        if(typeof(SP) == Kprocess)
        {
            retrievefirstSEQIDLVALUE(op1.Vcollect->objects,obj);
            result = value_to_runstack(obj);
        }
        else
        {
        result.Vcollect = Ncollect;
        get_structure(&strc,allstrucs,stindex);
        foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
        {
            /* collect puts objects in structure struc that
                    are of same type as obj */
            collect_(obj,strc,&(result.Vcollect->objects));
            untouch(strc); /* collect marks nodes - must unmark */
        }
        }

        stack(result,&runstack);
        break;

case dotOp:

        op1 = pop(&runstack);

        /* get index into string array */
        ithinSEQInteger(body,code_pos++,lo);
        ithinSEQInteger(body,code_pos,hi);
        index = (hi << 8) | lo;

        /* retrieve value */
        ithinSEQString(string_refs,index,str_value);


    result.Vcollect = Ncollect;
    foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
    {
        /* get all attributes with same name as dot name */
            foreachinSEQattrDesc(obj.VnodeDesc->attributes,Sad,ad)
            {
                if(streq(str_value,ad->name))
                    {
                        appendrearSEQIDLVALUE(result.Vcollect->
                                                        objects,ad->value);
                    }
            }

    }

        code_pos--; /* decrement for accurate runtime_result entry */
    stack(result,&runstack);
        code_pos = code_pos + 2;        /* adjust to next operator */

            break;

case typeExpressionOp:
            /* get index into type array */
            ithinSEQInteger(body,code_pos++,lo);
            ithinSEQInteger(body,code_pos,hi);
            index = (hi << 8) | lo;

            /* retrieve value */
            ithinSEQtypeTree(type_refs,index,type_value);
            result.Vcollect = Ncollect;

            /* get all types reachable from given type */
            /* (type may be a class node) */
            reach(type_value,&types);
    foreachinSETString(types,Ss,s)
    {
            /* convert string name of type to equivalent object */
       string_to_object(&obj,s,type_value);
            get_structure(&strc,allstrucs,stindex);
            /* collect objects in structure strc of same type
               as obj */
            collect_(obj,strc,&(result.Vcollect->objects));
            untouch(strc); /* collect marks nodes - must unmark */
    }

     code_pos--; /* decrement for accurate runtime_result
                             entry */
     stack(result,&runstack);
     code_pos = code_pos + 2; /* adjust to next operator */
     break;

case portExpressionOp:

    /* get index into type array */
    ithinSEQInteger(body,code_pos++,lo);
    ithinSEQInteger(body,code_pos++,hi);
    index = (hi << 8) | lo;

    /* retrieve value */
    ithinSEQtypeTree(type_refs,index,type_value);
    result.Vcollect = Ncollect;

    /* get all types reachable from given type */
    /* (type may be a class node)  */
    reach(type_value,&types);

    /* get index into string array */
    ithinSEQInteger(body,code_pos++,lo);
    ithinSEQInteger(body,code_pos,hi);
    index = (hi << 8) | lo;

    /* retrieve value */
    ithinSEQString(string_refs,index,str_value);
    /* str_value is port name */

    foreachinSETString(types,Ss,s)
    {
            /* convert string name of type to equivalent object */
            string_to_object(&obj,s,type_value);
            get_structure_port(&strc,str_value,allstrucs,numstruc);
            /* collect objects in structure struc of same type
               as obj */
            collect_(obj,strc,&(result.Vcollect->objects));
            untouch(strc); /* collect marks nodes - must unmark */
    }

     code_pos = code_pos - 3; /* decrement for accurate
                                         runtime_result entry */
     stack(result,&runstack);
     code_pos = code_pos + 4; /* adjust to next operator */
     break;

 case forallOp:
    op1 = pop(&runstack);

    /* store iterator collect in new object so that
            original won't get clobbered */
    op2.Vcollect = Ncollect;
    foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
       appendfrontSEQIDLVALUE(op2.Vcollect->objects,obj);
```

```
          /* get first object to iterate with */
          retrievefirstSEQIDLVALUE(op2.Vcollect->objects,obj);
          removefirstSEQIDLVALUE(op2.Vcollect->objects);

          /* place object into control array */
          controlArray[nestlevel++] = obj;
          numcontrol++;
          /* place rest of collection set on quantifier
                 collections list */
          appendrearSEQcollect(quantsets,op2.Vcollect);
          break;

      case existsOp:

          op1 = pop(&runstack);

          /* store iterator collect in new object so that
                 original won't get clobbered */
          op2.Vcollect = Ncollect;
          foreachinSEQIDLVALUE(op1.Vcollect->objects,Sobj,obj)
             appendfrontSEQIDLVALUE(op2.Vcollect->objects,obj);

          /* get first object to iterate with */
          retrievefirstSEQIDLVALUE(op2.Vcollect->objects,obj);
          removefirstSEQIDLVALUE(op2.Vcollect->objects);

          /* place object into control array */
          controlArray[nestlevel++] = obj;
          numcontrol++;
          appendrearSEQcollect(quantsets,op2.Vcollect);

          break;

      case endForAll:
          /* get return point */
          ithinSEQInteger(body,code_pos,entry);

          op1 = pop(&runstack);
          if(typeof(op1) == KFVALUE) /* body is false! */
          {
                  removelastSEQcollect(quantsets);
                  result.VFVALUE = NFVALUE;
                  stack(result,&runstack);

                  /* place object that caused failure in result array */
                  runtime_result[code_pos] =
                     value_to_runstack(controlArray[nestlevel-1]);
                  nestlevel--;
                  code_pos++;
          }
          else /* body is true */
          {
                  retrievelastSEQcollect(quantsets,qs);
                  /* if not more to check, quantifier is true */
                  if(emptySEQIDLVALUE(qs->objects))  /* set is empty */
                  {
                      removelastSEQcollect(quantsets);
                      result.VTVALUE = NTVALUE;
                      stack(result,&runstack);
                      nestlevel--;
                      code_pos++;
                  }
                  else /* go get next in set */
                  {
                      retrievefirstSEQIDLVALUE(qs->objects,obj);
                      removefirstSEQIDLVALUE(qs->objects);
                      controlArray[nestlevel-1] = obj;
```

```
                code_pos = entry;
            }
        }
        break;

    case endExists:

        ithinSEQInteger(body,code_pos,entry);
        op1 = pop(&runstack);
        if(typeof(op1) == KTVALUE) /* body is true! */
        {
                removelastSEQcollect(quantsets);
                result.VTVALUE = NTVALUE;
                stack(result,&runstack);
                /* place object that satisfied Exists in result array */
                runtime_result[code_pos] =
                    value_to_runstack(controlArray[nestlevel-1]);
                nestlevel--;
                code_pos++;
        }
        else /* body is false */
        {
                retrievelastSEQcollect(quantsets,qs);
                /* if not more to check, quantifier is false */
                if(emptySEQIDLVALUE(qs->objects))  /* set is empty */
                {
                    removelastSEQcollect(quantsets);
                    result.VTVALUE = NTVALUE;
                    result.VFVALUE = NFVALUE;
                    stack(result,&runstack);
                    nestlevel--;
                }
                else /* go back to get next in set */
                {
                    retrievefirstSEQIDLVALUE(qs->objects,obj);
                    removefirstSEQIDLVALUE(qs->objects);
                    controlArray[nestlevel-1] = obj;

                    code_pos = entry;

                }
        }
        break;

    case controlOp:
       result.Vcollect = Ncollect;
       /* get level */
       ithinSEQInteger(body,code_pos,level);

       /* get offset into control array */
       retrievefirstSEQInteger(controlReturn,offset);
       /* retrieve object */
       obj = controlArray[level+offset-1];
       appendrearSEQIDLVALUE(result.Vcollect->objects,obj);

       stack(result,&runstack);
       code_pos++;
       break;

       case formArgOp:

           result.Vcollect = Ncollect;
           /* get position of actual argument */
           ithinSEQInteger(body,code_pos++,pos);

           /* retrieve actual argument */
           arg = forms[pos-1];
```

```
                        appendrearSEQIDLVALUE(result.Vcollect->objects,arg);

                        stack(result,&runstack);
                        break;


                    case applicationOp:

                            /* get index into definition array */
                        ithinSEQInteger(body,code_pos++,lo);
                        ithinSEQInteger(body,code_pos++,hi);
                            index = (hi << 8) | lo;

                            /* retrieve value */
                            ithinSEQDefOrInst(define_refs,index,def_value);

                            /* get number of arguments */
                        ithinSEQInteger(body,code_pos++,numargs);
                            args = NULL;
                            /* get arguments */
                        for(i=numargs-1;i>=0;i--)
                            {
                              argrun = pop(&runstack);
                              if(typeof(argrun) == Kcollect)
                                  argrun = get_object(argrun);
                              arg = runstack_to_value(argrun);
                              /* place in array to pass to interpreter */
                              formalArray[i] = arg;
                              /* place in sequence to pass to routine that
                                 finds instance that matches arguments */
                              appendfrontSEQIDLVALUE(args,arg);
                            }

                            /* structures to handle recursion */
                            appendfrontSEQInteger(controlReturn,numcontrol);
                            numcontrol = 0;

                            if(typeof(def_value) == KidlInstance)
                                instance = def_value.VidlInstance;
                            else /* must find instance */
                          instance = findinstance(def_value,args);

                            result = interpret(instance->postfixDefn,
                                               name,SP,spos,formalArray,
                                               allstrucs,numstruc,stindex,tree);

                            code_pos = code_pos - 3;
                            stack(result,&runstack);
                            code_pos = code_pos + 3;
                            removefirstSEQInteger(controlReturn);
                            break;


        } /* end of switch */

        /* get next code operator */
        ithinSEQInteger(body,code_pos++,entry);
} /* end of while loop */

result = pop(&runstack);
if(entry == RETURN)          /* just interpreted a definition body -
                                        return result */
{
        return(result);
}
else /* interpreted an assertion body */
```

```
   {
K(Assertion completed);
              /* Find out whether assertion is true and report */
              if(typeof(result) == KFVALUE)
              {
                     if(typeof(name) == KTvoid)
                     {
                       if(typeof(SP) == Kstructure)
                        fprintf(stderr,"Assertion in %s is false.0,
                                         allstrucs[stindex].portname);
                       else
                        fprintf(stderr,"Assertion is false.0);

                     if(!suppress)
                       print(tree,0,FALSE,0); /* generate error log */
                     }
                     else
                     {
                     if(typeof(SP) == Kstructure)
                      fprintf(stderr,"Assertion %s in %s is false.0,
                               name.VnameTok
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *                                                                       *
 *  Title:  Supporting Routines for Assertion Checker          *
 *  Filename:       ~kickenso/softlab/check/check2.c                *
 *  Author:         Jerry Kickenson <kickenso@unc>                       *
 *                  Department of Computer Science                       *
 *                  University of North Carolina                  *
 *                  Chapel Hill, NC  27514                               *
 *                                                                       *
 *  Copyright (C) The University of North Carolina, 1985        *
 *                                                                       *
 *  All rights reserved. No part of this software may be sold or  *
 *  distributed in any form or by any means without the prior written   *
 *  permission of the SoftLab Software Distribution Coordinator. *
 *                                                                       *
 *  Report problems to          softlab@unc (csnet) or                   *
 *                              softlab!unc@CSNET-RELAY (ARPAnet)  *
 *  Direct all inquiries to the SoftLab Software Distribution     *
 *          Coordinator, at the above addresses.                  *
 *                                                                       *
 *  Function: Supporting routines for interpreter for assertion checker  *
 *                                                                       *
 *                                                                       *
 ******************************************************************* */

static char rcsid = "$Header: check2.c,v 1.3 85/12/27 14:15:57 kickenso Exp $"


#include <stdio.h>
#include "Check.h"
#include "macros.h"
#include "../codegen/instructions.h"



/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *          stack_obj  stack a runstack entry on the runstack and  *
 *                          place entry in given position in result array *
 *                                                                       *
 *          Last revised:       April 14, 1986                           *
 ***********************************************************************/

stack_obj(obj,stk,pos)
runstackEntry       obj;                /* object to stack */
SEQrunstackEntry    *stk;       /* runstack*/
int                 pos;                /* position in result array to place object */
{
   extern runstackEntry         runtime_result[];

   appendfrontSEQrunstackEntry(*stk,obj);
   runtime_result[pos] = obj;
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *          pop         pop the runstack                           *
 *                                                                       *
 *          Last revised:       April 14, 1986                           *
 ***********************************************************************/

runstackEntry               pop(stk)
SEQrunstackEntry    *stk;       /* runstack */
{
   runstackEntry    retval;

   retrievefirstSEQrunstackEntry(*stk,retval);
   removefirstSEQrunstackEntry(*stk);
   return(retval);
```

```
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           len         len returns the length of a generic list          *
*                                                                         *
*           Last revised:       April 14, 1986                            *
**********************************************************************/

len(list)
pGenList  list;         /* list to get length of */
{
   int                  length;
   pGenList             listptr;

   listptr = list;
   length = 0;

   while(listptr != NULL)
   {
          length++;
          listptr = listptr->next;
   }

   return(length);

}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           reach       place in the given set strings representing all types   *
*                       reachable from the given type.  Used to expand class     *
*                       types into their constituent node types.                 *
*                                                                                *
*           Last revised:       June 7, 1986                                     *
**********************************************************************/

reach(t,incl)
typeTree  t;            /* type that does the reaching */
SETString *incl;        /* set in which to place type representations */
{

   SEQNT                Snt;
   NT                   nt;
   SETString            nt_reach();/* reach for nonterminals */


   *incl = NULL;

   switch (typeof(t)) {
          case Kbool:
             addSETString(*incl,BOOL);
             break;

          case KTint:
             addSETString(*incl,INT);
             break;

          case Krat:
             addSETString(*incl,RAT);
             break;

          case Kstr:
             addSETString(*incl,STR);
             break;

          case Kset:
             addSETString(*incl,SET);
```

```
                break;

            case Kseq:
                addSETString(*incl,SEQ);
                break;

            case Ksingleton:
                reach(t.Vsingleton->object_type,incl);
                break;

            case Kuser:
                if(typeof(t.Vuser->NT) != KClassNT)
                        {addSETString(*incl,t.Vuser->NT.IDLclassCommon->name);}
                    else
                    /* class type - trace down nonterminal descendants */
                foreachinSEQNT(t.Vuser->NT.VClassNT->descendants,Snt,nt)
                            *incl = nt_reach(nt,*incl);
                break;

            }
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           nt_reach   places in the given set the string           *
*                      representations of types reachable through the    *
*                      given nonterminal.                                *
*                                                                          *
*           Last revised:      April 14, 1986                              *
*********************************************************************/

SETString nt_reach(n,i)
NT                n;         /* nonterminal doing the reaching */
SETString i;           /* set in which to place type represetations */
{
            SETString strset;
            SEQNT              Snt;
            NT                 nt;

            if(len( (pGenList)i) == 0)
                strset = NULL;
            else
                strset = i;

            if(typeof(n) != KClassNT)
                {addSETString(strset,n.IDLclassCommon->name);}
            else
              foreachinSEQNT(n.VClassNT->descendants,Snt,nt)
                  strset = nt_reach(nt,strset);
            return(strset);
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           collect            place in the given collection all objects in  *
*                              given structure of the same type as the given     *
*                              object.                                              *
*                                                                                    *
*           Last revised:      April 14, 1986                                        *
*********************************************************************/

collect_(obj,data,collects)
IDLVALUE           obj;        /* get objects with same type as this object */
nodeDesc data;       /* structure */
SEQIDLVALUE        *collects; /* collection in which to place found objects */
{

    SEQattrDesc                Sad;
```

```
 attrDesc                    ad;
 SETIDLVALUE                 Ssetval;
 IDLVALUE                    val;
 SEQIDLVALUE                 Sseqval;

 if(typeof(obj) != KnodeDesc) /* object is a basic type or set or sequence */
 {
         if(NodeTouched(data) == 0)
         {
                     MarkTouched(data);
                     foreachinSEQattrDesc(data->attributes,Sad,ad)
                     {
                                 if(typeof(ad->value) == typeof(obj))
                                 {
                                     if(typeof(obj) == KsetDesc)
                                     /* get only sets with same component type */
                                     {
                                             if(set_subset(ad->value.VsetDesc->value,
                                                          obj.VsetDesc->value))
                                                 appendrearSEQIDLVALUE((*collects),
                                                                                 ad->value);
                                     }
                                     else
                                     if(typeof(obj) == KsequenceDesc)
                                     /* get only sequences with same component type */
                                     {
                                             if(seq_subset(ad->value.VsequenceDesc->value,
                                                          obj.VsequenceDesc->value))
                                                 appendrearSEQIDLVALUE((*collects),
                                                                                 ad->value);
                                     }
                                     else
                                         appendrearSEQIDLVALUE((*collects),ad->value);
                                 }
                                 else
                                 /* follow path down structure */
                                     if(typeof(ad->value) == KnodeDesc)
                                         collect_(obj,ad->value.VnodeDesc,collects);
                     }
         }
 }
 else /* object is a node */
 {
 /* if node has same name, place in collection */
         if(NodeTouched(data) == 0)
         {
             MarkTouched(data);
             if(streq(obj.VnodeDesc->name,data->name))
                 appendrearSEQIDLVALUE((*collects),data);

         /* follow nodes down structure */
             foreachinSEQattrDesc(data->attributes,Sad,ad)
             {
                 if(typeof(ad->value) == KnodeDesc)
                         collect_(obj,ad->value.VnodeDesc,collects);
                     else
                     {
                         if(typeof(ad->value) == KsetDesc)
                         {
                                 foreachinSETIDLVALUE(ad->value.VsetDesc->value,
                                                                 Ssetval,val)
                                     if(typeof(val) == KnodeDesc)
                                             collect_(obj,val.VnodeDesc,collects);
                         }
                         else
                         {
                                 if(typeof(ad->value) == KsequenceDesc)
```

```
                                        foreachinSEQIDLVALUE(ad->value.VsequenceDesc->value,
                                                                   Sseqval,val)
                                             if(typeof(val) == KnodeDesc)
                                                collect_(obj,val.VnodeDesc,collects);
                              }
                           }
                     }
                  }
   }

   untouch(data);      /* unmark nodes marked by procedure collect */
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          untouch              unmark all nodes in given structure            *
*                                                                                *
*          Last revised:        April 14, 1986                                  *
**********************************************************************/

untouch(node)
nodeDesc  node;      /* structure to be ummarked */
{

   SEQattrDesc                 Sad;
   attrDesc                    ad;
   SETIDLVALUE                 Setidl;
   SEQIDLVALUE                 Seqidl;
   IDLVALUE                    idl;


   if(NodeTouched(node) != 0)
   {
           UnmarkTouched(node);
           foreachinSEQattrDesc(node->attributes,Sad,ad)
           {
              if(typeof(ad->value) == KnodeDesc)
                 untouch(ad->value.VnodeDesc);
           }
   }
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          findinstance         return the instance of the given definition   *
*                               which matches the given arguments              *
*                                                                                *
*          Last revised:        April 14, 1986                                  *
**********************************************************************/

idlInstance findinstance(def,args)
definition  def;       /* definition */
SEQIDLVALUE      args;      /* arguments to match */
{
   idlInstance                 retval;
   int                         OK;
   int                         i;
   Instance                    in;
   SETInstance                 Sin;
   formal           f;
   runstackEntry    a;


   foreachinSETInstance(def.IDLclassCommon->overload,Sin,in)
           if(len((pGenList)args) == len((pGenList)(in.IDLclassCommon->formals)))
           {
              OK = TRUE;
              /* check that each argument matches the formal by type */
```

```
            for(i=1;i<=len((pGenList)args);i++)
            {
                  ithinSEQformal(in.IDLclassCommon->formals,i,f);
                  ithinSEQIDLVALUE(args,i,a);
                  if(!type_match(a,f))
                     OK = FALSE;
            }
            if(OK == TRUE)
                  retval = in.VidlInstance;
         }

   return(retval);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*        type_match          return TRUE if the type of the given argument      *
*                            matches the type of the given formal               *
*                                                                                 *
*        Last revised:       April 14, 1986                                       *
*******************************************************************/

type_match(arg,form)
IDLVALUE          arg;        /* argument */
formal            form;       /* formal   */
{
   booleanretval;
   String  argtype;
   String  translate();/* string equivalent of IDL type */
   SETString        Ss;
   String  s;
   SETString        incl;
   SETString        incl1,incl2;
   SETIDLVALUE  So;
   IDLVALUE         o;


   incl1 = NULL;
   incl2 = NULL;
   retval = TRUE;
   argtype = translate(arg);

         /* types match if type of argument is included in the set of all
            types reachable by the formal type */

         switch (typeof(arg))  {
            case KsetDesc:
                  if(typeof(form->type) != Kset)
                     retval = FALSE;
                  else /* component types must match */
                  {
                     reach(form->type.Vset->component,&incl1);
                     foreachinSETIDLVALUE(arg.VsetDesc->value,So,o)
                           addSETString(incl2,translate(o));
                     if(!includes(incl1,incl2))
                           retval = FALSE;
                  }
                  break;
            case KsequenceDesc:
                  if(typeof(form->type) != Kseq)
                     retval = FALSE;
                  else /* component types must match */
                  {
                     reach(form->type.Vseq->component,&incl1);
                     foreachinSEQIDLVALUE(arg.VsequenceDesc->value,So,o)
                           addSETString(incl2,translate(o));
                     if(!includes(incl1,incl2))
                           retval = FALSE;
```

```
                    }
                    break;
            default:
                    reach(form->type,&incl);
                    if(!inSETString(incl,argtype))
                        retval = FALSE;
                    break;
        }

    return(retval);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          includes   returns true if the first argument includes    *
*                            all members of the second argument               *
*                                                                             *
*          Last revised:          April 14, 1986                              *
***************************************************************************/

includes(s1,s2)
SETString s1,s2;
{

          boolean                retval;
          SETString Ss;
          String                 s;

          retval = TRUE;
          foreachinSETString(s2,Ss,s)
             if(!inSETString(s1,s))
                    retval = FALSE;

          return(retval);
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          translate   returns the string equivalent of the IDL type        *
*                                                                             *
*          Last revised:          April 14, 1986                              *
***************************************************************************/

String     translate(arg)
IDLVALUE            arg;        /* argument whose type we want translated */
{

   String  result = NULL;
   char    *malloc();

   switch (typeof(arg))  {
          case KbooleanDesc:
             result = malloc(5);
             strcpy(result,BOOL);
             break;
          case KintegerDesc:
             result = malloc(4);
             strcpy(result,INT);
             break;
          case KstringDesc:
             result = malloc(4);
             strcpy(result,STR);
             break;
          case KrationalDesc:
             result = malloc(4);
             strcpy(result,RAT);
             break;
            case KsetDesc:
                result = malloc(4);
                strcpy(result,SET);
                break;
            case KsequenceDesc:
                result = malloc(4);
                strcpy(result,SEQ);
                break;
            case KnodeDesc:
                result = malloc(strlen(arg.VnodeDesc->name)+1);
                strcpy(result,arg.VnodeDesc->name);
                break;
            default:
                break;
            }
            return(result);
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           get_object returns the the object the given collection   *
*                               contains.  The collection must be a singleton      *
*                               collection.                                        *
*                                                                                   *
*           Last revised:       April 14, 1986                                      *
*********************************************************************/

runstackEntry           get_object(coll)
runstackEntry           coll;       /* collection to get object from  */
{
            /* coll is known to be a singleton collection */

            IDLVALUE            temp;
            runstackEntry       result;

            /* put object into temp */
            retrievefirstSEQIDLVALUE(coll.Vcollect->objects,temp);

            /* coerce collection into that object */
            switch (typeof(temp)) {
               case KintegerDesc:
                        result.VintegerDesc = NintegerDesc;
                        result.VintegerDesc->value = temp.VintegerDesc->value;
                        break;
               case KrationalDesc:
                        result.VrationalDesc = NrationalDesc;
                        result.VrationalDesc->value = temp.VrationalDesc->value;
                        break;
               case KbooleanDesc:
                        result.VbooleanDesc  = NbooleanDesc;
                        result.VbooleanDesc->value = temp.VbooleanDesc->value;
                        break;
               case KstringDesc:
                        result.VstringDesc = NstringDesc;
                        result.VstringDesc->value = temp.VstringDesc->value;
                        break;
               case KsetDesc:
                        result.VsetDesc = NsetDesc;
                        result.VsetDesc->value = temp.VsetDesc->value;
                        break;
               case KsequenceDesc:
                        result.VsequenceDesc = NsequenceDesc;
                        result.VsequenceDesc->value = temp.VsequenceDesc->value;
                        break;
               case KnodeDesc:
                        result.VnodeDesc = NnodeDesc;
                result.VnodeDesc->name = temp.VnodeDesc->name;
                result.VnodeDesc->attributes = temp.VnodeDesc->attributes;
                result.VnodeDesc->label = temp.VnodeDesc->label;
                break;
        }

        return(result);
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       rustack_to_value        returns the runstack entry equivalent        *
*                                       to the given IDLVALUE                   *
*                                                                               *
*       Last revised:           April 14, 1986                                  *
*********************************************************************/

IDLVALUE        runstack_to_value(obj)
runstackEntry   obj;            /* object to get rustack equivalent of */
{

        IDLVALUE        retval;

        switch (typeof(obj))  {
          case KintegerDesc:
                retval.VintegerDesc = obj.VintegerDesc;
                break;
          case KrationalDesc:
                retval.VrationalDesc = obj.VrationalDesc;
                break;
          case KbooleanDesc:
                retval.VbooleanDesc = obj.VbooleanDesc;
                break;
          case KstringDesc:
                retval.VstringDesc = obj.VstringDesc;
                break;
          case KsetDesc:
                retval.VsetDesc = obj.VsetDesc;
                break;
          case KsequenceDesc:
                retval.VsequenceDesc = obj.VsequenceDesc;
                break;
          case KnodeDesc:
                retval.VnodeDesc = obj.VnodeDesc;
                break;
          case KTVALUE:
                retval.VbooleanDesc = NbooleanDesc;
                retval.VbooleanDesc->value = TRUE;
                break;
          case KFVALUE:
                retval.VbooleanDesc = NbooleanDesc;
                retval.VbooleanDesc->value = FALSE;
                break;
          default:/* collections cannot be function arguments */
                break;
        }

        return(retval);

}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       value_to_runstack       returns the IDLVALUE equivalent of the       *
*                                       given runstack entry            *
*                                                                               *
*       Last revised:           April 14, 1986                                  *
```

```
***********************************************************************/

runstackEntry         value_to_runstack(val)
IDLVALUE              val;
{

            runstackEntry         retval;

            switch (typeof(val))  {
               case KintegerDesc:
                        retval.VintegerDesc = val.VintegerDesc;
                        break;
               case KrationalDesc:
                        retval.VrationalDesc = val.VrationalDesc;
                        break;
               case KbooleanDesc:
                        retval.VbooleanDesc = val.VbooleanDesc;
                        break;
               case KstringDesc:
                        retval.VstringDesc = val.VstringDesc;
                        break;
               case KsetDesc:
                        retval.VsetDesc = val.VsetDesc;
                        break;
               case KsequenceDesc:
                        retval.VsequenceDesc = val.VsequenceDesc;
                        break;
               case KnodeDesc:
                        retval.VnodeDesc = val.VnodeDesc;
                        break;
            }

            return(retval);

}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           set_subset returns TRUE if the first argument is a subset         *
*                               of the second                                   *
*                                                                               *
*           Last revised:       April 14, 1986                                  *
***********************************************************************/

set_subset(set1,set2)
SETIDLVALUE        set1,set2;
{
            boolean               result;
            SETIDLVALUE           Sval;
            IDLVALUE              val;
            SETString Sname;
            String                name;
            SETString incl1 = NULL;
            SETString incl2 = NULL;
            String                translate();



            result = TRUE;

            foreachinSETIDLVALUE(set1,Sval,val)
               addSETString(incl1,translate(val));
            foreachinSETIDLVALUE(set2,Sval,val)
               addSETString(incl2,translate(val));

            foreachinSETString(incl1,Sname,name)
               if(!inSETString(incl2,name))
                        result = FALSE;
```

```
        return(result);
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       seq_subset returns TRUE if the first argument is a subset      *
*                       of the second                                       *
*                                                                           *
*       Last revised:       April 14, 1986                                  *
**********************************************************************/
seq_subset(seq1,seq2)
SEQIDLVALUE     seq1,seq2;
{
        boolean             result;
        SEQIDLVALUE         Sval;
        IDLVALUE            val;
        SETString Sname;
        String              name;
        SETString incl1 = NULL;
        SETString incl2 = NULL;
        String              translate();


        result = TRUE;

        foreachinSEQIDLVALUE(seq1,Sval,val)
        {
           name = translate(val);
           addSETString(incl1,name);
        }

        foreachinSEQIDLVALUE(seq2,Sval,val)
        {
           name = translate(val);
           addSETString(incl2,name);
        }

        foreachinSETString(incl1,Sname,name)
           if(!inSETString(incl2,name))
                   result = FALSE;
        return(result);
}



/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*       string_to_object    passes back the IDLVALUE object      *
*                                   equivalent to the type represented by     *
*                                   the given string.                         *
*                                                                           *
*       Last revised:       April 14, 1986                                  *
**********************************************************************/

string_to_object(obj,s,tval)
IDLVALUE            *obj;       /* object passed back through here */
String              s;          /* string representation of type   */
typeTree   tval;        /* actual type - needed for set/sequence types */
{

        IDLVALUE            val;
        SETString St;
        String              t;
        SETString comp_types;           /* component types of set/sequence */


        if(streq(s,BOOL))
           (*obj).VbooleanDesc = NbooleanDesc;
```

```
            else
             if(streq(s,INT))
                 (*obj).VintegerDesc = NintegerDesc;
            else
             if(streq(s,RAT))
                 (*obj).VrationalDesc = NrationalDesc;
            else
             if(streq(s,STR))
                 (*obj).VstringDesc = NstringDesc;
            else
             if(streq(s,SET))
             /* must put appropriate component types into object */
             {
                 (*obj).VsetDesc = NsetDesc;
                 reach(tval.Vset->component,&comp_types);
                 foreachinSETString(comp_types,St,t)
                 {
                         string_to_object(&val,t,tval);
                         addSETIDLVALUE((*obj).VsetDesc->value,val);
                 }
             }
            else
             if(streq(s,SEQ))
             /* must put appropriate component types int object */
             {
                 (*obj).VsequenceDesc = NsequenceDesc;
                 reach(tval.Vset->component,&comp_types);
                 foreachinSETString(comp_types,St,t)
                 {
                         string_to_object(&val,t,tval);
                         appendrearSEQIDLVALUE((*obj).VsequenceDesc->value,val);
                 }
             }
            else
              {
                (*obj).VnodeDesc = NnodeDesc;
                (*obj).VnodeDesc->name = s;
              }
}

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          get_structure          pass back the structure at the position given*
*                                 by the index.  Structure is found in          *
*                                 the given list of structures.                 *
*                                                                                    *
*          Last revised:          April 17,1986                                      *
*********************************************************************/

get_structure(stc,allstrucs,index)
nodeDesc  *stc;         /* structure to pass back */
struc                   allstrucs[]; /* list of all structure instances */
int                     index;      /* index of structure in allstrucs */
{
                        *stc = allstrucs[index].st;
}



/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          get_structure_port     pass back the structure associated          *
*                                          with the given port.  Structure is *
*                                          found in the given list of structures      *
*                                                                                    *
*          Last revised:          April 17, 1986                                     *
*********************************************************************/

get_structure_port(stc,name,allstrucs,num)
```

```
nodeDesc  *stc;         /* structure to pass back */
String                  name;     /* name of port */
struc                   allstrucs[]; /* list of all structure instances */
int                     num;      /* number of structures in allstrucs          */
{

          int           i;

          for(i=0;i<=num;i++)
              if(streq(allstrucs[i].portname,name))
                        *stc = allstrucs[i].st;

}
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *                                                                   *
 *  Title:  Error Log Printer for Assertion Checker          *
 *  Filename:       ~kickenso/softlab/check/errorprint.c             *
 *  Author:         Jerry Kickenson <kickenso@unc>                    *
 *                  Department of Computer Science                   *
 *                  University of North Carolina                     *
 *                  Chapel Hill, NC  27514                            *
 *                                                                   *
 *  Copyright (C) The University of North Carolina, 1985     *
 *                                                                   *
 *  All rights reserved. No part of this software may be sold or  *
 *  distributed in any form or by any means without the prior written   *
 *  permission of the SoftLab Software Distribution Coordinator. *
 *                                                                   *
 *  Report problems to          softlab@unc (csnet) or               *
 *                              softlab!unc@CSNET-RELAY (ARPAnet)    *
 *  Direct all inquiries to the SoftLab Software Distribution        *
 *          Coordinator, at the above addresses.                     *
 *                                                                   *
 *  Function: Prints error log for unsatisfied assertion on stout    *
 *                                                                   *
 *                                                                   *
 ****************************************************************** */


#include <stdio.h>
#include "Check.h"
#include "macros.h"


#define NONBOOL  -1

#define write0(str)             fprintf(stderr,str)
#define write1(str,s1)          fprintf(stderr,str,s1)
#define write2(str,s1,s2)       fprintf(stderr,str,s1,s2)


extern runstackEntry  runtime_result[1000];

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 *          print       print an error log reporting values of intermediate    *
 *                      expressions in the given expression tree.        *
 *                                                                       *
 *          Last revised:         June 7,1986                            *
 *********************************************************************/

print(exp,TF,ind,head)
expression exp;         /* expression to be printed      */
boolean                 TF;     /* print only if True(T) or False(F)      */
int                     ind; /* indentation level       */
int                     head;   /* header - 0 is '>'(because), 1 is '&' (and)*/
{


            String      nameof();
            String      entry_to_string();
            String      type_to_string();
            int         pos;    /* position in runtime result array         */
            int         boolvalue; /* boolean value of result      */
            char        temp_str[80];
            int         p;
            boolean     found;
            SEQexpressionPair   Sep;
            expressionPair                  ep;
```

```
/* get position in result array where result of expression is */
pos = exp.IDLclassCommon->valuepos + 1;

if(typeof(runtime_result[pos]) == KTVALUE)
          boolvalue = TRUE;
else if(typeof(runtime_result[pos]) == KFVALUE)
          boolvalue = FALSE;
else
   boolvalue = NONBOOL;


switch (typeof(exp))  {

          case Kbinary:
          /* if value of expression is boolean, print only if it
             agrees with given truth value (TF)          */

                    if(boolvalue == FALSE && boolvalue == TF)
                    {
                              indent(ind);
                              header(head);
                              write1("%s not satisfied.0,
                                        nameof(exp.Vbinary->op));
                              print(exp.Vbinary->left,FALSE,ind+1,0);
                              if(typeof(runtime_result[exp.Vbinary->
                                 left.IDLclassCommon->valuepos + 1])
                                 == KTVALUE)
                                 print(exp.Vbinary->right,FALSE,ind+1,0);
                              else
                                 print(exp.Vbinary->right,FALSE,ind+1,1);

                    }
                    else if(boolvalue == TRUE && boolvalue == TF)
                    {
                              indent(ind);
                              header(head);
                              write1("%s is satisfied.0,
                                        nameof(exp.Vbinary->op));
                              print(exp.Vbinary->left,TRUE,ind+1,0);
                              if(typeof(runtime_result[exp.Vbinary->
                                 left.IDLclassCommon->valuepos + 1])
                                 == KFVALUE)
                                 print(exp.Vbinary->right,TRUE,ind+1,0);
                              else
                                 print(exp.Vbinary->right,TRUE,ind+1,1);
                    }
                    else if(boolvalue == NONBOOL)
                    /* print all non-boolean expressions */
                    {
                              indent(ind);
                              header(head);
                              strcpy(temp_str,entry_to_string(
                                        runtime_result[pos]));
                              write2("%s is %s.0,nameof(exp.Vbinary->op),
                                        temp_str);
                              print(exp.Vbinary->left,TRUE,ind+1,0);
                              print(exp.Vbinary->right,TRUE,ind+1,1);
                    }
                    break;

          case Kunary:

                    /* if boolean Not, print if value agrees with
                       passed truth value (TF)          */
                    if(typeof(exp.Vunary->op) == Knot)
                    {
                    if(boolvalue == TF)
```

```
                {
                    indent(ind);
                    header(head);
                            if(boolvalue == FALSE)
                            {
                                write0("NOT is not satisfied.0);
                                print(exp.Vunary->body,TRUE,ind+1,0);
                            }
                            else        /* boolvalue = TRUE */
                            {
                                write0("NOT is satisfied.0);
                                print(exp.Vunary->body,FALSE,ind+1,0);
                            }
                }
                }
                else /* print non-boolean minus always */
                {
                            indent(ind);
                            header(head);
                            strcpy(temp_str,entry_to_string(
                                        runtime_result[pos]));
                            write1("INVERSE is %s.0,temp_str);
                            print(exp.Vunary->body,TRUE,ind+1,0);
                }
                break;

        case Kdotted:

                indent(ind);
                header(head);
                /* get attribute name */
                strcpy(temp_str,entry_to_string(runtime_result[pos]));
                write2(".%s is %s0,exp.Vdotted->right->name,
                            temp_str);
                print(exp.Vdotted->left,TRUE,ind+1,0);
                break;

        case KtypeExpression:

                indent(ind);
                header(head);
                /* get type name */
                strcpy(temp_str,entry_to_string(runtime_result[pos]));
                write2("%s is %s0,type_to_string(exp.VtypeExpression
                            ->type),temp_str);

                break;

        case KdefnRef:

                indent(ind);
                header(head);
                /* get result of application */
                strcpy(temp_str,entry_to_string(runtime_result[pos]));
                write2("%s returns %s0,exp.VdefnRef->name,temp_str);
                break;

        case Kroot:

                indent(ind);
                header(head);
                strcpy(temp_str,entry_to_string(runtime_result[pos]));
                write1("ROOT is %s0,temp_str);
                break;

        case Kmembers:
              indent(ind);
              header(head);
              strcpy(temp_str,entry_to_string(runtime_result[pos]));
              write1("MEMBERS are %s0,temp_str);
              print(exp.Vmembers->argument,TRUE,ind+1,0);
              break;

      case Khead:

              indent(ind);
              header(head);
              strcpy(temp_str,entry_to_string(runtime_result[pos]));
              write1("HEAD is %s0,temp_str);
              print(exp.Vhead->argument,TRUE,ind+1,0);
              break;

      case Ktype:

              indent(ind);
              header(head);
              strcpy(temp_str,entry_to_string(runtime_result[pos]));
              write1("TYPE is %s0,temp_str);
              print(exp.Vtype->argument,TRUE,ind+1,0);
              break;

      case Ksize:

              indent(ind);
              header(head);
              strcpy(temp_str,entry_to_string(runtime_result[pos]));
              write1("SIZE is %s0,temp_str);
              print(exp.Vsize->argument,TRUE,ind+1,0);
              break;

      case Ktail:

              indent(ind);
              header(head);
              strcpy(temp_str,entry_to_string(runtime_result[pos]));
              write1("TAIL is %s0,temp_str);
              print(exp.Vtail->argument,TRUE,ind+1,0);
              break;

      case Kquantifier:

              /* print expression only if value agrees with
                 passed truth value (TF) */
              if(boolvalue == TF && boolvalue == FALSE)
              {
                 indent(ind);
                 header(head);
                 if(typeof(exp.Vquantifier->op) == Kforall)
                 {
                    /* get name of object for which ForAll failed */
                    strcpy(temp_str,entry_to_string(
                                 runtime_result[pos+1]));
                    write2("FORALL is not satisfied when %s is %s0,
                                 temp_str);
                          print(exp.Vquantifier->set,TRUE,ind+1,0);
                          print(exp.Vquantifier->body,FALSE,ind+1,1);
                 }
                 else  /* Exists operator        */
                          write0("EXISTS is not satisfied.0);
              }

              if(boolvalue == TF && boolvalue == TRUE)
              {
```

```
                        indent(ind);
                        header(head);
                        if(typeof(exp.Vquantifier->op) == Kforall)
                        {
                                write0("FORALL is satisfied.0);
                        }
                        else
                        {
                          /* get name of object for which Exists was
                                  satisfied */
                          strcpy(temp_str,entry_to_string(
                                          runtime_result[pos+1]));
                                write2("EXISTS is satisfied when %s is %s0,
                                          exp.Vquantifier->control->name,
                                          temp_str);
                                print(exp.Vquantifier->set,TRUE,ind+1,0);
                                print(exp.Vquantifier->body,TRUE,ind+1,1);
                        }
                  }

                  break;

        case Kconditional:

                  indent(ind);
                  header(head);
                  if(boolvalue == TRUE && boolvalue == TF)
                  {
                     write0("Conditional is satisfied.0);
                  }
                  else if((boolvalue == FALSE && boolvalue == TF) ||
                             (boolvalue == NONBOOL))
                  {
                     write0("Conditional is not satisfied.0);
                     p = exp.Vconditional->test.IDLclassCommon
                                                   ->valuepos;
                     if(typeof(runtime_result[p+1]) == KTVALUE)
                 {
                         print(exp.Vconditional->test,TRUE,ind+1,0);
                         print(exp.Vconditional->then,FALSE,ind+1,1);
                 }
                 else
                 {
                         found = FALSE;
                         foreachinSEQexpressionPair(
                                 exp.Vconditional->orif, Sep, ep) {
                         if(!found)
                         {
                                   p = ep->test.IDLclassCommon->valuepos;
                                   if(typeof(runtime_result[p+1]) == KTVALUE)
                                   {
                                     print(ep->test,TRUE,ind+1,0);
                                     print(ep->then,FALSE,ind+1,1);
                                     found = TRUE;
                                   }
                               }
                               }
                      }

                      if(!found)
                               print(exp.Vconditional->otherwise,
                                                   TRUE,ind+1,0);
                  }

                  break;

        case Kcontrol:
```

```
                                        indent(ind);
                                        header(head);
                                        strcpy(temp_str,entry_to_string(runtime_result[pos]));
                                        write2("%s is %s0,exp.Vcontrol->name,temp_str);
                                        break;

                        case KformArg:

                                        indent(ind);
                                        header(head);
                                        strcpy(temp_str,entry_to_string(runtime_result[pos]));
                                        write2("%s is %s0,exp.VformArg->name,temp_str);
                                        break;

            }

}



/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           indent                  indent the given amount of levels.  One level       *
*                                   is 4 spaces.                                            *
*                                                                                           *
*           Last revised:           April 14, 1986                                          *
*********************************************************************/

indent(x)
int         x;          /* indent level */
{

            int         i;

            for(i=1;i<=4*x;i++)
                        fprintf(stderr," ");
}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           header                  print the symbol '>' (because) or '&' (and) *
*                                   depending on whether the given argument is        *
*                                   0 (>) or 1 (&)                                          *
*                                                                                           *
*           Last revised:           April 14, 1986                                          *
*********************************************************************/

header(head)
int         head;       /* header code */
{

            if(head == 0)
                        fprintf(stderr,"> ");
            else
                        fprintf(stderr,"& ");
}

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*           nameof                  return the string equivalent of the given     *
*                                   operator                                            *
*                                                                                           *
*           Last revised:           April 14, 1986                                          *
*********************************************************************/

String      nameof(op)
binaryOp    op;         /* operator to get string equivalent of */
```

```
{

        switch (typeof(op))   {

                case Kand:
                        return("And");
                        break;
                case Kor:
                        return("Or");
                        break;
                case KTunion:
                        return("Union");
                        break;
                case Kintersect:
                        return("Intersection");
                        break;
                case Kplus:
                        return("Sum");
                        break;
                case Kminus:
                        return("Difference");
                        break;
                case Ktimes:
                        return("Product");
                        break;
                case Kdivide:
                        return("Quotient");
                        break;
                case Kless:
                        return("Less");
                        break;
                case KlessEq:
                        return("Less_Eq");
                        break;
                case Kgreater:
                        return("Greater");
                        break;
                case KgrtrEq:
                        return("Greater_Eq");
                        break;
                case Kequal:
                        return("Equality");
                        break;
                case KnotEqual:
                        return("InEquality");
                        break;
                case Ksame:
                        return("Same");
                        break;
                case KinSet:
                        return("Set Inclusion");
                        break;
                case Ksubset:
                        return("Subset");
                        break;
                case KpropSubset:
                        return("Proper Subset");
                        break;

        }

}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*          entry_to_string                    returns a string equivalent of the *
*                                             given runstack entry              *
*                                                                                *
*          Last revised:        April 14, 1986                                   *
*********************************************************************************/

String     entry_to_string(entry)
runstackEntry          entry;      /* runstack entry to get string equivalent of */
{
           char                 temp[80];
           IDLVALUE             val;
           SEQIDLVALUE          Sval;
           int                  i;
           int                  n;

           /* fill temp array with Null characters */
           for(i=0;i<=79;i++)
              temp[i] = ' ';

           switch  (typeof(entry))           {

                      case KintegerDesc:
                                 sprintf(temp,"%d",entry.VintegerDesc->value);
                                 break;
                      case KrationalDesc:
                                 sprintf(temp,"%f",entry.VrationalDesc->value);
                                 break;
                      case KstringDesc:
                                 strcpy(temp,entry.VstringDesc->value);
                                 break;
                      case KbooleanDesc:
                                 if(entry.VbooleanDesc->value == TRUE)
                                            strcpy(temp,"TRUE");
                                 else
                                            strcpy(temp,"FALSE");
                                 break;
                      case KnodeDesc:
                                 strcpy(temp,entry.VnodeDesc->label);
                                 break;
                      case KsetDesc:
                                 strcpy(temp,"{ ");
                                 foreachinSETIDLVALUE(entry.VsetDesc->value,Sval,val)
                                 {
                                            strcat(temp,entry_to_string(val));
                                            strcat(temp," ");
                                 }
                                 strcat(temp,"}");
                                 break;
                      case KsequenceDesc:
                                 strcpy(temp,"< ");
                                 foreachinSEQIDLVALUE(entry.VsequenceDesc->value,Sval,
                                                                                  val)
                                 {
                                            strcat(temp,entry_to_string(val));
                                            strcat(temp," ");
                                 }
                                 strcat(temp,">");
                                 break;
                      case KTVALUE:
                                 strcpy(temp,"TRUE");
                                 break;
                      case KFVALUE:
                                 strcpy(temp,"FALSE");
                                 break;
                      case Kcollect:
                                 strcpy(temp,"{ ");
                                 n = 0;
```

```
                                        foreachinSEQIDLVALUE(entry.Vcollect->objects,
                                                                        Sval,val)
                                        {
                                            n++;
                                            if(n <= 15)
                                            {
                                                    strcat(temp,entry_to_string(val));
                                                    strcat(temp," ");
                                            }
                                        }
                                        if(n < len((pGenList)(entry.Vcollect->objects)))
                                            strcat(temp,"...");
                                        strcat(temp,"}");
                                        break;

                }

                return(temp);

}


/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*               type_to_string                  return a string equivalent of the *
*                                               given type                       *
*                                                                                 *
*               Last revised:           April 14, 1986                            *
*******************************************************************/

String          type_to_string(t)
typeTree        t;              /* type to get string equivalent of */
{


                switch (typeof(t))      {

                        case KTint:
                                        return("INTEGER");
                        case Krat:
                                        return("RATIONAL");
                        case Kstr:
                                        return("STRING");
                        case Kbool:
                                        return("BOOLEAN");
                        case Kset:
                                        return("SET");
                        case Kseq:
                                        return("SEQ");
                        case Kuser:
                                        return(t.Vuser->DT->name);

                }

}
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*                                                                     *
*  Title:  Interpretation of Assertions                         *
*  Filename:        ~kickenso/softlab/seman/macros.h                  *
*  Author:          Jerry Kickenson <kickenso@UNC>                    *
*                   Department of Computer Science                    *
*                   University of North Carolina                      *
*                   Chapel Hill, NC  27514                            *
*                                                                     *
*  Copyright (C) The University of North Carolina, 1985         *
*                                                                     *
*  All rights reserved. No part of this software may be sold or  *
*  distributed in any form or by any means without the prior written  *
*  permission of the SoftLab Software Distribution Coordinator. *
*                                                                     *
*  Report problems to          softlab@unc (csnet) or                 *
*                              softlab!unc@CSNET-RELAY (ARPAnet)      *
*  Direct all inquiries to the SoftLab Software Distribution      *
*        Coordinator, at the above addresses.                         *
*                                                                     *
*  Function: macros used by all phases of assertion interpretation    *
*                                                                     *
*                                                                     *
****************************************************************** */


#include <stdio.h>

#define K(str)                  fprintf(stderr,"str 0);
#define check(X,N,SP,POS,TREE)         interpret(X,N,SP,POS,forms,allstrucs,                              numstruc,stind

#define streq(s1,s2)    strcmp(s1,s2) == 0
#define error(str)      if(pass >= 2) fprintf(stderr,str)

typedef int boolean;

typedef struct {                /* structure instance */
            nodeDesc  st;       /* pointer to root of structure instance */
            String  portname;  /* name of port associated with structure */
            String  stname;    /* name of structure */
        }           struc;

#define BOOL        "bool"
#define INT         "int"
#define RAT         "rat"
#define STR         "str"
#define SET         "set"
#define SEQ         "seq"

#define HEAD        "Head"
#define MEMBERS     "Members"
#define TYPE        "Type"
#define SIZE        "Size"
#define TAIL        "Tail"

#define UNKNOWN "Unknown"

#define strequal(s1,s2)         !strcmp(s1,s2)

#define COMMENT                 -1
#define EXTENSION-2
#define WARNING                 -3
#define RECOVERABLE             -4
#define SERIOUS                 -5
#define FATAL                   -6
```

```
#define Comment0(n,spos)              ErrHandler(COMMENT,n,spos,"","")
#define Comment1(n,spos,a1)           ErrHandler(COMMENT,n,spos,a1,"")
#define Comment2(n,spos,a1,a2)        ErrHandler(COMMENT,n,spos,a1,a2)

#define Extension0(n,spos)            ErrHandler(EXTENSION,n,spos,"","")
#define Extension1(n,spos,a1)         ErrHandler(EXTENSION,n,spos,a1,"")
#define Extension2(n,spos,a1,a2) ErrHandler(EXTENSION,n,spos,a1,a2)

#define Warning0(n,spos)              ErrHandler(WARNING,n,spos,"","")
#define Warning1(n,spos,a1)           ErrHandler(WARNING,n,spos,a1,"")
#define Warning2(n,spos,a1,a2)        ErrHandler(WARNING,n,spos,a1,a2)

#define Recoverable0(n,spos)          ErrHandler(RECOVERABLE,n,spos,"","")
#define Recoverable1(n,spos,a1)       ErrHandler(RECOVERABLE,n,spos,a1,"")
#define Recoverable2(n,spos,a1,a2)    ErrHandler(RECOVERABLE,n,spos,a1,a2)

#define Serious0(n,spos)              ErrHandler(SERIOUS,n,spos,"","")
#define Serious1(n,spos,a1)           ErrHandler(SERIOUS,n,spos,a1,"")
#define Serious2(n,spos,a1,a2)        ErrHandler(SERIOUS,n,spos,a1,a2)

#define Fatal0(n,spos)                ErrHandler(FATAL,n,spos,"","")
#define Fatal1(n,spos,a1)             ErrHandler(FATAL,n,spos,a1,"")
#define Fatal2(n,spos,a1,a2)          ErrHandler(FATAL,n,spos,a1,a2)
```