

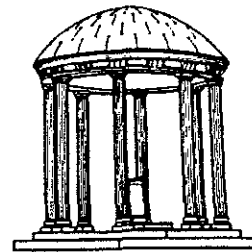
LLPT: A Little-Language Prototyping Tool

TR89-037

October, 1989

J. Christopher Ramming

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



UNC is an Equal Opportunity/Affirmative Action Institution.

LLPT: A Little-Language Prototyping Tool

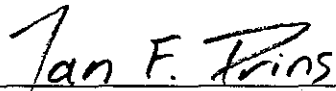
by
J. Christopher Ramming

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

1989

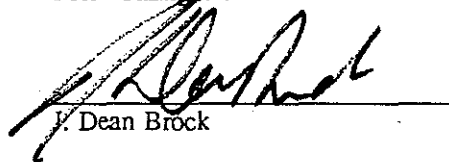
Approved by:



Jan Prins



Peter Calingaert



J. Dean Brock

JAMES CHRISTOPHER RAMMING
LLPT: A Little-Language Prototyping Tool
(Under the direction of Jan Prins)

ABSTRACT

Application-specific languages can offer productivity boosts under the appropriate circumstances, but there is an impediment to their proliferation: language implementation is inherently difficult. Interest in a tool for rapidly prototyping these "little languages" motivates this thesis. It identifies three specific problem areas that offer undue hindrance to language prototypers. These are (1) concrete syntax specification, (2) abstract parse tree construction, and (3) the definition of a language's semantics. This thesis describes conventional approaches to these issues and explains why they are inappropriate for the language prototyper. A simplified approach to each problem is developed and a unified tool implementing these techniques is described.

Table of Contents

Chapter I: Introduction	1
Language Implementation	1
Problems with Language Implementation	1
Chapter II: Three Improvements	4
Full Grammar Specification	4
Concrete-to-Abstract Syntax Translation	11
Semantic Synthesis	18
Summary	22
Chapter III: Description of LLPT	23
Grammar Specification	23
Tree Construction	28
Semantic Actions	31
Summary	35
Chapter IV: Conclusion	36
Advantages	36
Drawbacks	36
Suggestions	37
References	39
Appendix A: A Desk Calculator	41
Appendix B: Code Listings	47

CHAPTER I INTRODUCTION

1. Language Implementation

Executing a program in some particular language involves two phases. First its syntactic structure must be examined to ensure that it has the correct form. If so, a semantic synthesis step must analyze this structure to produce results prescribed by the program. Hence language implementation involves the construction of components that perform these tasks.

The syntactic structure of the language in question should be formally defined with conveniences like Backus-Naur Form (BNF) grammars or regular expressions. A parser that embodies this syntactic definition must be crafted to read input programs and determine whether they are in the specified form. If so, a translator must map this input to an internal representation that can be consulted during subsequent analysis.

The second phase, called semantic synthesis, operates on the representation of a syntactically valid program as determined by the previous phase. The semantics of the language must be defined either on an ad-hoc basis or with the help of certain formalisms like those of attribute grammars and denotational semantics. This definition is based on the syntactic structure of input programs and either yields an execution of the program or produces a program in yet another language. If the yield is an execution, the term "interpreter" describes the system; otherwise, the term "translator" is used. Collectively this thesis will refer to these as language processors.

2. Problems with Language Implementation

This thesis begins with a look at various methods of syntactic analysis. The traditional approach distributes this task between a scanner and parser. This approach has much to commend it, but it also has implications that are unpleasant for the language prototyper. Because unified grammar specifications appear to resolve these problems, previous systems aimed at this goal are discussed. But each of these

is shown to have problems of its own. The most promising approach has been to permit a unified grammar specification with an extended BNF from which both a scanner and parser can be generated; this thesis develops certain refinements to this approach.

Next, the problem of constructing an internal representation of source programs is examined. This representation traditionally takes the form of a rooted graph called the abstract parse tree, and is the result of a translation step that maps concrete syntax to abstract syntax. This translation has previously been performed in an ad-hoc way during parsing (with the use of reduction-time actions) or it has been managed with the help of an expert system. But both methods seems too inconvenient and complex, so this thesis develops conventions and BNF extensions that permit this translation to be specified at a high level.

Finally, the problem of semantic synthesis is attacked. It is fair to say that most language processors indulge in ad-hoc semantic synthesis. This is because no formal approaches have gained the universal approval and support that their counterparts in syntactic analysis enjoy. But there are some disciplines that can assist the language prototyper. Two of these are attribute grammars and denotational semantics. The former involves attaching attributes and equations to grammar rules, and includes systems which walk parse trees to find attribute values by evaluating equations. Denotational semantics is predicated on the assumption that the semantics of a language are most clearly and easily defined by specifying (with highly mathematical notation) an interpreter for the language. Its approach to interpreter specification, viz., by semantic functions tied directly to abstract grammar productions, seems to be a nice principle for interpreter organization.

This thesis observes that an object-oriented approach to programming could assist those who intend to follow (at least loosely) the disciplines of either attribute grammars or denotational semantics. Thus a flexible but suggestive framework for interpreter implementation is presented: tree nodes will be instances of classes in some object-oriented language, and a special notation will permit the user to associate attribute fields and semantic functions directly with grammar productions.

This thesis makes several contributions. It develops a convenient method for specifying a language's syntax so that a parser and scanner can be constructed automatically. It shows how the concrete-to-abstract syntax translation can be specified at a high level so that this too can take place automatically. Finally, it develops a framework that assists in the semantic synthesis phase. These are concrete suggestions, and in order to see their effect on a real system they are implemented in an experimental tool called LLPT, which stands for the "Little-Language Prototyping Tool". This tool is described in the final portions of the thesis.

Remark: the reference to "little" languages deserves some comment. LLPT will not be targeted toward the prototyping of arbitrary languages, because a narrower scope will be used to justify decisions that introduce restrictions but result in greater convenience. "Little languages" are relatively simple, from both user and machine perspectives; it is this simplicity that LLPT trades on.

CHAPTER II THREE IMPROVEMENTS

This section of the thesis identifies three problems which are common to language prototyping and have heretofore made this task difficult and error-prone. (1) The task of syntactic analysis is typically distributed over a scanner and parser (requiring the two to cooperate carefully). (2) Concrete grammars do not correspond to abstract grammars (making automatic parse tree construction difficult). (3) Assigning semantics to a language is still an art, not a science (making it difficult to provide support that is convenient without being restrictive). Work that addresses these difficulties will be analyzed, and specific suggestions will be made for improving the language prototyper's lot.

1. Full Grammar Specification

1.1. Motivation

The language recognition subtask has traditionally been broken up into two major pieces: the language scanner and the language parser. Input to a scanner consists of a stream of characters. The scanner takes these characters and assembles them into tokens. A stream of these tokens is then presented to a component which parses them according to some grammar.

Assuming this division of labor, one must use a certain amount of discretion to fix on the appropriate granularity of scanning. At one end of the spectrum lies the trivial scanner, which recognizes each character as a token. This leaves all of the language recognition task to the parser component. The other end of the spectrum is that in which the scanner is a parser in its own right, with the ability to recognize an entire program as a token. This implies a degenerate parser.

This division of tasks is organized around the following rule of thumb. A complete (character-based) grammar for a language is imagined or designed. Those nonterminals deriving language components that could be described conveniently with regular expressions are declared as tokens. The scanner is then implemented as a finite automaton. This pruning has left the designer with a grammar which requires a more powerful recognition mechanism, such as a pushdown automaton.

There are a number of good things to be said about this plan related to the fact that notation and automata for regular languages are simpler than those for the more powerful context-free languages. Regular expressions are more compact and easier to use (because of the many R.E. operators) than grammars, and if one had to implement the underlying machines one would find that grammars require more complex, slower, and larger programs. Thus a scheme which permits one to use either regular expressions or grammars may result in clearer, faster, smaller recognition mechanisms.

Also, by making the distinction between parser and scanner it is possible to do things that are difficult to describe grammatically. For instance, it is hard and tedious to account for whitespace with an LALR(1) grammar, yet easy for the scanner simply to strip it away before attempting to recognize tokens.

Finally, this division provides an opportunity to describe languages that are not even context-free. In C[1], for instance, the `typedef` facility changes the token class of a given string. Unfortunately, this feature takes C out of the class of context-free languages (intuitively, this can be seen by noting that a pushdown automaton has no place to store these new keywords for later recall), even though it is easy to accommodate by adding state to the scanner.

But if the scanner and parser are developed separately a developer must manage the interface between them carefully. Maintaining this interface is an onerous burden to the prototyper on three counts. There is redundancy because there must be a way for the scanner and parser to communicate with meaning about the tokens that are recognized, and some convention must therefore be embodied in each tool. In addition, as changes to the language are made they must be reflected in both the scanner and the parser. Finally, the decision as to scanner granularity is itself hard. One would prefer to retain the advantages of a separate scanner without succumbing to the problem of managing such an interface.

1.2. Scanner/Parser Partnerships.

Scanner and parser generators exist and are generally based on finite automata and pushdown automata, respectively. Lex[2] and yacc[3] are two of the most popular tools for these tasks, and do a nice

job in their respective fields. One of their most helpful features is that they are designed to work well together. For instance, lex produces a scanner that can be invoked as the procedure `yylex()`; not coincidentally, this is the precisely the function that yacc assumes will invoke the scanner.

Lex and yacc represent substantial advances in the technology of syntactic analysis. These tools contain much knowledge about their special functions, the creation of language recognition automata. Furthermore, they provide convenient interfaces to this expertise. Thus it is no longer necessary to understand how such automata are created: only the ability to specify them is required.

Aficionados of lex and yacc might say that the problem of redundancy is well worth the flexibility that results from the separation of tasks, and they would argue that these tools minimize redundancy by virtue of their design. Nonetheless, it will be shown that one can do better.

1.3. Scannerless Parsing

A paper which recognizes the problem of redundancy and tries to resolve it is presented by Salomon[4], who proposes that language recognizers be specified without scanners: that is, the grammar is to be fully specified on a character level. Although this seems a simple matter, two kinds of ambiguity are hard to resolve with context-free grammars. These are called "reserved-identifier" ambiguities and "longest-match" ambiguities. Salomon's solution is to permit these ambiguities in a typical BNF grammar and then to restrict the language generated with two new directives: "exclusion" rules and "adjacency-restriction" rules.

The reserved-identifier problem involves confusion due to the fact that keywords are frequently a subset of identifiers. If the parser uses a scanner, then the scanner generally distinguishes between these by consulting a table. But how should scannerless parsers accomplish that task? Salomon's solution is to extend the BNF with an "exclusion rule". This rule can be used to say that the language generated by some symbol does not contain the language generated by another symbol. For example, the exclusion rule

```
id ::= begin | end | if | while ;
```

indicates that certain keywords are not identifiers.

In addition, Salomon introduces an adjacency-restriction rule. This rule is intended to resolve the kinds of ambiguity that result from, for instance, keywords embedded in identifiers. It works by making a statement about which symbols are forbidden to occupy adjacent positions. Here is an example of the rule:

```
identifier if else ... -/- identifier if else ...
```

Items that could be generated by the symbols to the left of `-/-` are not permitted next to items that could be generated by items on the right. Thus, this example indicates that an identifier may not abut any keywords or other identifiers.

But as expected, this solution has a number of problems. First, whitespace is still hard to account for grammatically because of its pervasive nature: in many languages it can appear virtually anywhere. Second, it becomes necessary to specify tokens using BNF rather than regular expressions. Third, it does not permit the description of languages that are not context free. Finally, experiments indicate that larger, slower parsers result from this method than from standard scanner/parser solutions.

Moreover, these BNF extensions do not provide any additional expressive power beyond that provided by, for instance, yacc and its disambiguating rules. The reserved-identifier problem is in fact just a reduce/reduce ambiguity, and yacc users are capable of solving by ordering the grammar appropriately. Likewise, the adjacency-restriction rule is introduced to resolve the kinds of shift/reduce conflicts that yacc users have been resolving for years. Thus Salomon goes to much effort to introduce extensions to BNF that do not solve any new problems and (at least in the case of the adjacency-restriction rule) have effects that are hard to understand.

Salomon's system solves the redundancy of separate parser/scanner specification without alleviating any of the problems that are inherent in a character-level specification of the grammar.

1.4. Implicit Scanners.

The best thing about scannerless parsing is that a language can be completely specified in one fell swoop. Other approaches try to maintain this single-specification advantage in different ways. One solution is to allow an extended BNF that replaces terminal identifiers with regular expressions describing a token. A scanner can then be constructed using these regular expressions. This approach is taken by a number of systems, including Eli[5] and FrEGe[6]. This approach has the advantages of a separate parser/scanner architecture without its deficiencies.

For instance, a description of C's while loops might look as follows (although the exact syntax varies from system to system):

```
whileloop -> "while" "(" expression ")" stmt ;
```

By examining such a grammar it is possible to determine what the various tokens are, and a scanner that recognizes these tokens can be systematically constructed.

This certainly solves the problem of redundancy in language specifications. Unfortunately, none of these tools is very sophisticated, and only simple languages can be described because of two major restrictions.

One restriction is that the scanners are constructed in a context-insensitive manner: that is, the question of which tokens may be scanned is not answered on the basis of the parser's state. Consider the task of writing a parser for the yacc language. Yacc programs consist of three different sections (declarations, rules, and functions). Each of these has a different grammatical structure, and it could be said that each has a different lexical environment as well, in the sense that one section may have keywords that are meaningless in another. For instance, the keyword `%token` is meaningful in the declaration section but not the rules section. Consequently, a scanner needs to behave with due regard for the environment in which it is invoked.

The second major restriction involves whitespace. These systems permit one to describe comments and whitespace but they are not general enough: languages with more than one whitespace

environment cannot be accounted for. Yet any language that includes string values, such as `sh` (the UNIX shell command language), must include rules for strings surrounded by double quotes. Because strings may contain metacharacters mixed with ordinary characters, it is necessary to parse them character-by-character. In parsing such strings the usual whitespace characters like blanks and tabs must be attended, although any backslash-newline combinations should be ignored! (Note that this example also exhibits another manifestation of the multiple-lexical-environment problem: parsers should not be tempted to recognize keywords within strings, even though they may well appear there).

The implicit scanner approach eliminates the redundancy inherent in two-level grammar specifications. Heretofore, however, scanner inference systems have not been flexible enough to accommodate multiple whitespace and lexical environments.

1.5. The LLPT Solution to Parser Specification

The challenge is to eliminate redundancy that arises from separate interfaces to a scanner and parser while simultaneously maintaining the advantages that result from this division of labor. To this end the solution of `Eli` and `FrEGe` is taken and various modifications are applied to eliminate difficulties.

The resulting scheme uses a BNF which has been extended in two ways: first, arbitrary regular expressions are permitted in place of terminal symbols, and second, each of these regular expressions is associated implicitly or explicitly with a whitespace environment. Analysis of a grammar presented in this form permits the inference of a scanner and a parser, and certain rules are used to resolve the ambiguities that arise from the use of regular expressions in the grammar.

The primary difference between an ordinary BNF grammar and our extended LLPT grammar is that where terminals appeared in the BNF grammar, regular expressions appear in the LLPT version. LLPT parser construction initially ignores the fact that terminals are regular expressions containing vital information; it constructs a parser on the assumption that a scanner will be built separately from this information.

The quality of the scanner is of concern to LLPT. Previously mentioned systems create relatively unintelligent scanners that do not alter their behavior on the basis of the parser's current state. This can always be done. But analysis of the parser permits the construction of much smarter scanners when parsers are implemented as automata that are driven from state on the basis of their input.

The controlling observation is that each of the parser's states can be associated with a "lexical context". A lexical context for some parser state is the set of tokens which do not drive the parser to an error state. If the parser is an automaton, then the information must be encoded somewhere and can be extracted. This information can then be used to construct a scanner which, when invoked, takes note of the parser's state and carefully recognizes only tokens in the corresponding lexical context. One way of thinking about this is to pretend that a different scanner is constructed for each lexical context.

LLPT will treat whitespace as follows. The user will be permitted to associate any terminal in the grammar with some notion of whitespace as described by a regular expression. The scanner will be constructed such that each invocation results in two operations (1) whitespace will be stripped away, and (2) an attempt will then be made to find a token which is part of the current lexical context. All tokens in the current lexical context should be associated with the same notion of whitespace; if this is not the case then error messages can be generated at compile time.

Ambiguity might well be introduced when the user embeds regular expressions in an LLPT grammar. It turns out that disallowing these ambiguities would cause the prototyper more harm than good, so they will be dealt with through the introduction of disambiguating rules. Naturally the user should be warned about the existence of any ambiguity in his specification.

One kind of possible ambiguity is that two tokens in the same lexical context may be regular expressions describing sets with a non-null intersection. This situation is not at all unlikely, and is in fact exhibited by the reserved-identifier problem mentioned earlier. If the scanner finds some string in this intersection it must decide which token to recognize. Some sort of disambiguating rule is needed here, and it will be chosen to mirror yacc's resolution of reduce/reduce conflicts. If two regular expres-

sions match the current input, the one that appears earlier in the grammar will be recognized.

A second situation involves lexical contexts in which there are two regular expressions with the property that a string could match one, but an even longer string could match the other (one regular expression includes some string that is a prefix of a string in the second regular expression). This is analogous to yacc's shift/reduce conflicts, and a similar disambiguating rule will be chosen: the longest possible string will always be matched.

These ambiguities do not pose an objection to the use of implied scanners. The reason for this is that any ambiguities discovered through such analysis must exist even if the scanner and parser are separately specified. By presenting them together one provides the generator with enough information to discover and report such ambiguities properly.

This scheme does not permit semantic actions to be associated with the recognition of tokens. This is a serious drawback in the sense that it makes LLPT an unattractive choice for implementing complicated (i.e. non-context-free) languages like C. However, little languages are not likely to call for such subtleties.

2. Concrete-to-Abstract Syntax Translation

2.1. Motivation

A language processor must do much more than simply decide whether its input is in the correct form. It must act on that input as well. Actions are conveniently expressed according to the structure of the input as defined by the grammar of the language. However, the parse according to the grammar is rarely the perfect platform for defining semantics, because many syntactic features exist only to impart structure to the token stream and have no intrinsic semantic value. Consequently there is no need to preserve them. In practice, an abstract parse tree is preferred to the concrete parse tree, because it elides superficial distinctions of form and in some instances has a different shape as well.

This section discusses previous work aimed at automatic parse tree construction, with reference to the way in which a language's abstract syntax is specified. A suggestion is then made as to how this

might be done in such a way as to please the language prototyper.

2.2. Ad-Hoc Construction

The usual way to produce a parse tree is to attach semantic actions to a grammar. These semantic actions contain descriptions, in some programming language, of what should be done to construct a tree. This method gives the programmer total flexibility over the tree that is constructed.

However, this flexibility is rarely called for. In the first place, code that describes how to build trees is monotonous: whenever there is a reduction (assuming a bottom-up parse) space is allocated for a node of the appropriate type, and then pointers to its children are installed in the node. One would certainly think that at least this much could be automated. In the second place, there is a small set of transformations that translate concrete syntax into abstract syntax (as will be seen later); thus only a few variations on tree building tend to be used in practice.

Another objection to ad-hoc tree construction is that the nature of the abstract grammar cannot be divined without poring through the various semantic actions attached to the grammar.

2.3. Systems with Specific Directives

A number of systems capitalize on the observation that the semantic actions which cause nodes (and, synthetically, trees) to be built are harder to understand than some notation which simply describes what these nodes should look like. This is true because certain typical actions can be inferred: space allocation and pointer assignment are good examples. Therefore, the reasoning goes, it would be nice if these descriptions could be used in place of the semantic actions. Two systems which follow this philosophy are y+[7] and FrEGe[5].

Y+ permits users to decorate a grammar rule with a Lisp-like expression that indicates how to build nodes of that type. Briefly, the notation uses braces, parentheses, and components of the rule. Any part of the rule may appear in the expressions, surrounded by either square brackets or parentheses. Square brackets around production items indicate that the enclosed items are to be siblings. Parentheses

around items indicate that the first item is to be the parent of the succeeding ones; if there is more than one item in the tail of the list, a sibling relationship is implied for these nodes. This notation is particularly good for eliminating unnecessary syntactic details from the parse tree and describing parent/sibling relationships. But this generality is somewhat unnecessary, as exceptionally complex relationships are unlikely to arise in practice. In any case, little languages should avoid this complexity.

FrEGe has a similar notation, with extensions involving keywords `@propagate`, `@list`, and `@unitlist`, amongst others. The first of these is a mechanism for chain rule elimination, and indicates that there is no node corresponding to the given production: rather, some value is to be propagated upward. `@list` and `@unitlist` are used to add children to an existing node (for instance if one is trying to create list nodes). These extensions provide certain useful flexibility above and beyond that of `y+`'s notation. In addition, FrEGe provides a default rule, which is an elementary but nonetheless substantial improvement over `y+`. The existence of a default rule suggests that there may in fact be a small number of "typical" node constructions; this observation will be useful later.

Using high-level directives to simplify the task of tree building has the advantage that these actions are much easier to understand and use than equivalent semantic actions written in a language like C. On the other hand, this method suffers from much the same criticism that can be applied to ad-hoc constructions: it is necessary to pore through what really amount to semantic actions in order to understand the form of abstract syntax trees. One has to question whether the flexibility of these systems really provides tremendous advantages; if one observes that unusual constructions rarely arise in practice, an even higher-level description may suffice for constructing parse trees.

2.4. Expert Systems

Bahrani has developed an interactive system called CAGT[8] which takes a concrete grammar and, with the user's help, identifies opportunities for simplification that yield a more convenient grammar. There are two major kinds of transformation. Chain rule elimination involves the elimination of rules like that of parentheses in arithmetic expression grammars. Similar-rule coalescing is a second kind of

transformation: CAGT identifies "similar" rules and factors out the differences, yielding a more manageable collection of rules.

The important aspect of CAGT is that it is an interactive expert system. Its input is a concrete grammar. Its output consists of a concrete grammar with appropriate semantic actions for building an abstract parse tree.

To a certain degree, CAGT relieves the programmer of the obligation to think. It dives into a concrete grammar, digests it, and then provides the programmer with a number of suggestions about the details of abstract syntax. The programmer signals yea or nay to each of these, and these responses are incorporated into a parser that automatically builds a tree.

However, this approach has the disadvantage that the user must be prepared to endure a session with the system each time a grammar is produced. This in and of itself is not so terrible, but as a consequence the relationship between concrete and abstract syntax is defined by the history of this session, which is a clumsy way to represent a relationship. One would prefer to define this relationship directly, perhaps with annotations to the grammar.

2.5. The LLPT Approach to Parse Trees

Clearly, many people have noted that parse trees should be built automatically by parsers. However, discrepancies between abstract and concrete syntax stand as a barrier to this goal. The nature of the abstract grammar has generally been clarified only by actual rules provided for tree construction (except in Bahrani's system). Some systems improve the situation by removing a level of detail from semantic actions that direct node construction, but the nature of the abstract grammar is still hidden in tree-building rules scattered throughout the grammar.

LLPT will build abstract parse trees automatically. However, the differences between its abstract and concrete grammars will be defined using certain conventions and very high-level decorations to the grammar. These will be sufficient for directing the process of tree construction. The observation which makes this possible is that there are a few "typical" node-building scenarios. It is sufficient to provide

for conventions and annotations that take these into account.

Of the various language features, arithmetic expressions seem to cause the most serious discrepancies between the abstract and concrete syntax trees. Here is an abstract grammar for a simple expression language:

```
expr: expr '-' expr | expr '**' expr | id ;
```

Even though this grammar expresses the form of a convenient abstract syntax tree, it is unsuitable for parsing on several grounds. It cannot be handled by a top-down parser (it is left-recursive and ambiguous) and it fails to enforce the common notions of precedence. A grammar which does not exhibit these unfortunate properties is contorted indeed. But happily there is no reason to worry about this particular case, because by using an LR parser with disambiguating rules both objections to the above rule are waved away! Bottom-up parsers like left-recursion, and clever use of disambiguating rules can cause ambiguous grammars to behave properly, even to the point of reflecting operator precedence. This demonstrates the importance of using the best parser generators available.

Thus it becomes necessary only to consider less serious differences between concrete and abstract syntax trees. The first involves the difficulty in specifying sequences, as in the following production:

```
stmtlist: /* empty */ | stmt | stmtlist stmt ;
```

the problem here is that this production has a parse tree which is tall and thin, when in fact one wants a single stmtlist node with any number of stmt's as its children. Therefore the BNF will be extended with the following shorthand for the above.

```
stmtlist: [ stmt ]* ;
```

When LLPT encounters an alternation enclosed by brackets and followed with a *, +, or ? it will interpret the rule as meaning zero-or-more, one-or-more, and zero-or-one of the entity within brackets, respectively. Note that the grammar is still specified unambiguously, but there are implications to the abstract syntax: the shape of the parse tree is understood differently as a result of this notation. Note: brackets are not permitted within a rule (these are not meant to be general regular-expression operators).

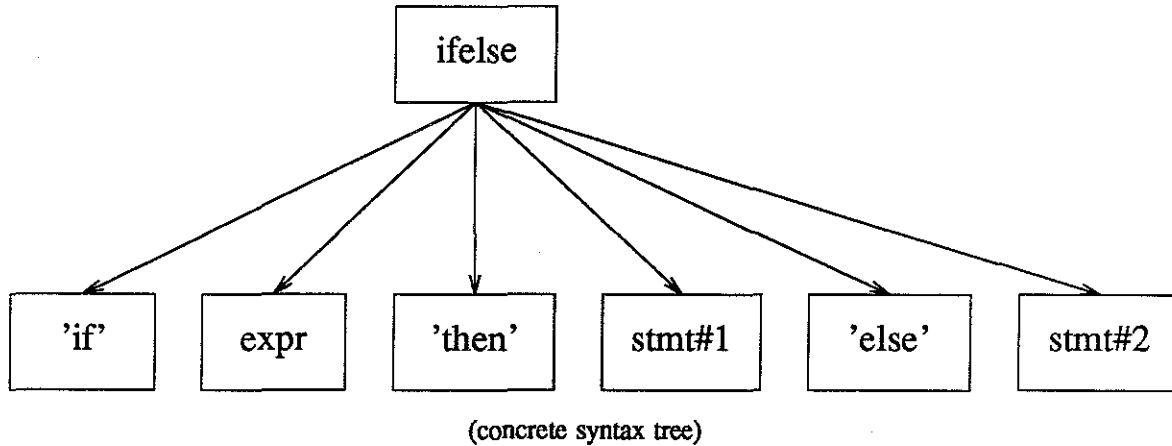
Remark: the standard BNF has only two operators, concatenation and alternation. Above, we have proposed that three extensions be made, with the restriction that they be used only on the top level. One might observe that if these extensions were made in a fully general way then the BNF might have the same economy of expression that characterizes regular expressions. Why, then, is this not done here? The reason is that syntactic analysis is not the final goal of the system. The parse tree that is implied by the grammar will actually be built and analyzed. Our intention is to assign meaning to productions by decorating them with user-definable attributes and semantic functions. This approach is convenient if productions are closely coupled to nodes in the parse tree. But general regular expression operators permit the implication of a parse tree in which certain nodes do not have corresponding productions. The restriction on these operators is imposed because it would be hard to find a convenient notation for coupling these anonymous nodes with functions and attributes.

A second difference between concrete and abstract syntax involves trivial leaves. A concrete syntax tree accounts for every terminal encountered during parsing: these become leaves of the tree. But many of these symbols are fixed strings (like `+` or `while`) that are useful only because they reveal which rule a node corresponds to. However, if each node is labeled explicitly these leaves become unnecessary as they can be recovered by implication from the name of the node. For example, an if-else rule can be described by this production:

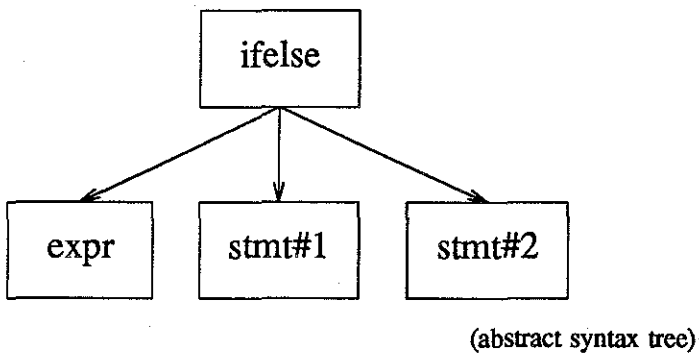
```
ifelse: 'if' expr 'then' stmt 'else' stmt
```

The concrete and abstract nodes for this production are as follows.

12.If 789



12.If 785



The terminals in the concrete node are superfluous, as they are the same for every instance of an 'ifelse' node. Therefore no fixed strings will be reflected in the parse tree; rather, each node will be labeled with an indication of which grammar rule caused it to be instantiated.

A third difference between concrete and abstract syntaxes involves the use of chain rules, which are rules in which the right-hand side of the production effectively involves a single symbol; one example of such a rule arises from arithmetic parenthesization.

expr: '(' expr ')'

one would not be happy to see a node corresponding to this rule in the abstract syntax tree because parentheses are only grouping operators without any semantic implications of their own.

LLPT will recognize any rule enclosed in angled brackets (< and >) to be a chain rule. No parse node will be constructed for such rules. Rather, the single nonterminal or complex terminal (one containing

regular expression metacharacters) on the right will be propagated upward.

These extensions to the standard BNF help somewhat in describing the nature of the abstract syntax; in any case, the most common uses of notation provided by FrEGe and y+ are accounted for. And in one case, there is a bonus in that the concrete syntax is more easily specified because of a notation that is reminiscent of regular expression operators.

Most important, however, is the fact that the concrete syntax of a language and its abstract syntax are clearly related. Because of this it will be possible to construct parse trees in a predictable way.

This scheme permits the user to describe the concrete syntax of a language while simultaneously specifying its abstract syntax. The unfortunate aspect of this is that in real life abstract syntax trees encode more syntactic information than can be reflected by context-free grammars. For instance, if a programming language includes variables the abstract syntax tree will most likely contain pointers from instances of a variable to the node in which it is declared. Because it is not possible to describe such things using CFG's there is no way for them to be reflected in our automatically built tree. Therefore, such information must be gained by walking the tree during the semantic phase.

3. Semantic Synthesis

3.1. Motivation

At some point in the compilation process it becomes necessary to determine whether the program is correct in ways that cannot be expressed using the syntactic specifications discussed up till now. In addition, if the program is a valid one, it will be necessary to imbue it with the appropriate interpretation. The language prototyper should be able to accomplish these things with a minimum of effort. Unfortunately, this is one of the least well-understood components of language processing. However, there are two formal approaches to the problem which should provide hints as to how it can be accomplished. Rather than present a new formalism for this aspect of the process, LLPT will encourage users to draw from the best of these by providing a model and some notation to assist in this process.

3.2. Attribute Grammars

One way to assign meaning to a program involves the use of attribute grammars. This approach considers that each node in the abstract syntax tree is associated with certain values called attributes. There are two kinds of attributes. Synthesized attributes are assigned a value on the basis of computation involving the attributes of a node's children. Inherited attributes are assigned a value on the basis of computation involving the attributes of a node's ancestors. These computations are described by equations. It is possible to determine which attributes are associated with a node by referring to the node's type. Likewise, the node's type determines the semantic equations that define the value of its attributes. (Two nodes with the same attribute may compute the value of that attribute with different semantic equations.)

One of the principal advantages of this approach is its declarative nature. One simply specifies the attributes of each node and the equations that can be used to determine their value. The task of walking through the parse tree to assign values to the attributes can be carried out automatically once the system of equations is known[9,10]. That this activity is implicit in attribute grammar systems is a boon to programmers.

But possibly more useful, for our purposes, is the fact that attribute grammars are a formal discipline for carrying out certain activities during the synthesis phase. The requirements of systematic approaches can often be anticipated, and support for them can be prepared in advance. We will see shortly how LLPT will support attribute grammars.

On the other hand, attribute grammars have their limitations. See [9,10,11] for more details on the theory and practice of attribute grammars.

3.3. Denotational Semantics

One of the problems with programming language semantics is that they are hard to define. In practice, compilers are frequently the best definitions of their languages' semantics. No matter how convoluted and hard to understand, they are improvements over other descriptions in that they are

completely unambiguous. Nonetheless, few compilers are error-free. And if a compiler can be said to have an error then there must be some ideal to which it is attempting to conform. Denotational semantics tries to define this ideal. At its heart lies the conviction that it ought to be easier to understand mathematically what a compiler ought to do than it is to define the operation of the program that implements it. This mathematical statement of the compiler's behavior is then to be taken as the language's semantic definition.

The idea behind denotational semantics is that the meaning of any given syntactic construct can be defined as a function of the elements which compose it and some notion of state; thus the meaning of a program (which is a syntactic unit) can be defined in terms of the semantics of those syntactic units that compose it, and so on in an inductive sort of way.

Thus a denotational semantics for some language is a collection of functions related to the syntactic units that compose the language. Note that by defining the denotational semantics of a program one is simply writing what is ordinarily recognized as an interpreter! The only difference between an "ordinary" interpreter and the denotational semantics of a language is that the latter utilizes highly mathematical notation and abstractions that could well be hard to implement efficiently.

A language processor can take the form of a translator, which generates code, or an interpreter, which executes it directly. The language prototyper should find it more convenient to specify an interpreter than a translator, for the reasons that (1) only the interpretation language and the source language need be mastered, whereas translators require the additional burden of treating a target language correctly, and (2) the result is something which can directly execute programs, eliminating an entire step from the write-compile-debug cycle common to software development. Thus there is a clear advantage to using denotational semantics for compiler prototyping.

Remark: some systems [6,12,13,14] are able to analyze the denotational semantics of a language, that is, an interpreter for it, in order to produce a translator for the language. Although interesting, this fact is irrelevant to compiler prototypers, as translation offers only improvements in time

efficiency.

Another advantage of denotational semantics is its systematic approach to interpreter construction, involving the association of semantic functions with syntactic units. This provides a convenient point of reference for the prototyper, who by following this recipe does not need to invent his own interpreter framework.

Finally, denotational semantics may not always be preferable to an operational semantics: some programming languages are convenient for operational definitions. McCarthy[15], for instance, had no qualms about using a meta-circular evaluator to define Lisp.

Whereas the mathematical notation of denotational semantics is an advantage in a descriptive sense, it is a severe drawback if there is no interpreter available for it: without such an interpreter one could never produce a working product by giving a denotational semantics for some language. In any case many mathematical constructs do not admit of practical implementation. Thus one would like to be able to take the formal approach of denotational semantics while stealing the practical advantages of a real-life language with which to define semantic functions and equations.

3.4. The LLPT Approach to Semantics

Because neither of the above methods is entirely appropriate in all situations, no attempt will be made to force LLPT users into either mold. But certain details of each method will be drawn upon to form a platform that supports the user, should either approach be found useful. The fact that both methods attach either code (in the form of equations and semantic function) or values (attributes) to rules in the abstract grammar suggests that LLPT's model and notational conveniences should assist this work. One way for LLPT to do this is to create automatically classes corresponding to grammar productions (using an object-oriented language). The nodes of the parse tree can then be instances of these classes, and synthesis can be accomplished by invoking the various methods associated with nodes.

An analysis of LLPT grammars will consequently result in the definition of C++[16] classes corresponding to each production. Further, a notation will permit the definition of semantic functions

and attributes that will be implemented as methods and fields of the class.

The class mechanism permits one to take advantage of denotational semantics' interpreter-construction methodology while simultaneously retaining an ability to perform activity much like that performed by attribute grammar systems. In addition, because C++ has mechanisms for defining abstract data types it will be possible for users to define classes that implement mathematical abstractions. Thus, if desired, the expressive power of mathematical abstractions can be introduced to facilitate the prototyper's job.

One drawback to this scheme, however, is that whereas it is easy to associate attributes with grammar rules and attach to a class methods which act as semantic equations for defining those attributes, LLPT will not provide a system which automatically walks the syntax tree to activate these functions. The user will have to manage that task explicitly. Another drawback is that whereas C++ provides a method for defining abstract data types, its users have a limited ability to define their own notation. Thus it is likely that the notation of denotational semantics will not be equaled.

4. Summary

The language implementation process is a difficult one. Certain traditional approaches to that task pose problems to those who are interested in rapid language prototyping. This thesis has now looked at three obstacles to that goal. It has found a way to simplify the problem of syntactic definition. It has found a way to describe an abstract grammar conveniently by decorating a concrete grammar. And it has found a way to encourage a structured approach to interpreter construction without being overbearing. It now remains to see how these have been implemented in a working system.

CHAPTER III DESCRIPTION OF LLPT

The major features of LLPT are a convenient grammar specification format, an automatic tree-building facility, and a mechanism for associating grammar productions with functions to perform the semantic synthesis. It is based heavily on yacc, and in fact is implemented largely by modifications to the standard yacc compiler. Many yacc features have been left in LLPT, and these will not be discussed in detail.

LLPT takes as input a program in the LLPT specification language. An LLPT program has three parts:

```
declarations (precedence, associativity, whitespace, C++)
%%
rules (grammar productions, semantic actions, etc).
%%
C++ functions
```

after examining this program, LLPT produces (1) a lexical analyzer, (2) a parser, and (3) a series of C++ class definitions corresponding to grammar productions. The parser recognizes and verifies input and also builds a parse tree from instantiations of the C++ classes corresponding to the rules recognized. These class definitions may well contain user-defined fields and member functions.

Thus the user ends up with a parser which, when invoked upon correct input, builds a tree automatically. If the user has defined the appropriate functions, it will be possible to walk the tree to perform type checking, evaluation, or code production as desired.

5. Grammar Specification

5.1. Implicit Scanners

LLPT users specify a language with a grammar that includes regular expressions wherever terminal symbols are desired. LLPT is able to analyze these grammars to create both a parser and an intelligent scanner. LLPT grammars specify an entire language, whereas ordinary grammars do not (because they elide token descriptions).

Here, for instance, is the specification of an LLPT parser for simple arithmetic expressions:

```
%left '+' /* Precedence, associativity */
%left '*' /* as in yacc... */
%%
expr : expr '+' expr
      | expr '*' expr
      | '(' expr ')'
      | integer
      ;
integer : "(+|-)?[0-9]+"
```

All rules are composed of nonterminals, literal terminals, and complex terminals. Quote marks enclose single quotes, and denote strings of characters. Complex terminals are surrounded by double quotes, and denote lex regular expressions rather than fixed strings. Note that the only difference between single and double quotes is that within the former lex regular expression meta-characters have no special significance, whereas in the latter they do. Any token not enclosed with single or double quotes is assumed to be a nonterminal, and must appear on the left of some rule.

LLPT initially treats such specifications as though the embedded regular expressions were simple terminals rather than notation with special significance. It then constructs an LALR(1) parser precisely as yacc itself would, reporting any ambiguities present on this level in the form of shift/reduce and reduce/reduce conflicts. These can be resolved as in yacc, by assigning precedences and defining the associativity of terminals.

After the parser has been constructed, LLPT examines both the parser and these regular expressions in order to create a scanner for the language. First, the LALR parse table is consulted. This table is an array with parser states along one dimension and tokens along the other. For each state, there is an indication of what should be done when a given token is encountered: shift, reduce, accept, or signal an error. Those tokens that do not cause an error are called the "lexical context" of that state. LLPT constructs context-sensitive scanners which consult the parser's state and look only for tokens in the corresponding lexical context.

Regular expressions describe sets, and it may happen that the input to the scanner matches more than one regular expression in the current lexical context. In this case, the scanner will choose the longest possible match. If the longest possible match is described by more than one expression, the scanner returns the token which was described earliest in the LLPT grammar. LLPT analyzes each lexical context and issues warning messages at compile-time whenever it finds the possibility of such ambiguity.

These disambiguating rules were chosen to mimic yacc's treatment of shift/reduce and reduce/reduce conflicts. However, because the latter rule may appear arbitrary, and since reordering the grammar may not seem a pleasing way to resolve the same-length ambiguities properly, a keyword `%tokprec` exists in LLPT which can be used to state the user's preference. It is used in the declaration section, and followed by the terminal expressions in the desired order as in this example.

```
%tokprec 'while', 'for', 'if', 'else', "[a-z]+"
```

This has the effect of stating that the usual keywords are to be returned even if there is a possible conflict with the regular expression `[a-z]+`, regardless of where these terminals appear in the subsequent grammar.

5.2. Whitespace Conventions

Scanners do more than recognize keywords: they are also used to strip away whitespace and comments. LLPT users can direct this activity by using the `%whitespace` keyword. This, like the `%tokprec` keyword, should be used in the declaration section of the program. It must be followed by a single regular expression which describes which characters should be stripped away before any attempt is made to match a token regular expression. Example:

```
%whitespace "([\ \n]|\(\.\))*"
```

will cause LLPT to ignore any combination of blanks, tabs, or C++ line-terminated comments.

There is also a provision for multiple definitions of whitespace within a single program. Suppose one wished to describe a C string with grammar rules. Such rules would look like this:

```

string: "" charlist ""
;
charlist: char
        | charlist char
        | /* empty */
;
char: "[\]"<STR> /* single characters other than " */
     | "\\"<STR> /* backslash escapes any character */
;

```

Ordinary whitespace is significant within strings, whereas backslash-newline combinations are not. In effect, tokens that make up C strings are associated with a non-standard kind of whitespace. This effect can be implemented by introducing the second kind of whitespace and giving it a nametag, say <STR>, that can be appended to tokens as required.

```

%whitespace "[ \n\t]*"
%whitespace<STR> "\n"
%%
string: "" charlist ""<STR>
;
charlist: char
        | charlist char
        | /* empty */
;
char: "[\]"<STR>
     | "\\"<STR>
;

```

This example provides an opportunity for a more precise explanation of whitespace semantics, which are as follows. In general, every input character is significant unless there is a global whitespace declared by use of the %whitespace keyword without a <name> suffix. Unless otherwise indicated, such global whitespace is stripped away before any attempt is made to match tokens. However, named whitespace expressions may exist, and if a terminal is tagged with a name then it is this kind of whitespace which is stripped away before the token can be recognized, rather than the usual kind. Thus in the modified rules above, the default whitespace is ignored before the first double quote, whereas only backslash-newline combinations are ignored before the second.

It is possible to tag an entire grammar rule with a whitespace name, as in

```

char<STR>: "[\]"
        | "\n"
        ;

```

In this case every terminal mentioned in the rule obeys the indicated convention. In fact, one should think of every rule-name as being associated with the default whitespace, even though there is no printable name for it.

However, one must be careful when using alternate whitespace conventions. The following demonstrates why.

```

% whitespace      "[ \t]"
% whitespace<NL> "\n"
%%
confusing: "\t"<NL>
        | "\n"
        ;

```

Under the default whitespace convention, newlines are significant. Under the NL convention, newlines are significant but tabs are not. Suppose there is an effort to match a string to the **confusing** rule: then both `\t<NL>` and `\n` are valid lookahead tokens for some parser state. But each uses a different whitespace convention. If the scanner permitted such things, it would not know upon input tab-newline whether to treat the tab as significant (under the `<NL>` convention) or skip over it (under the default convention). Since there is no obvious answer to this question, the situation is not permitted; it generates an error. The rule is that all tokens in a given lexical context must follow the same whitespace convention.

Note that the semantics of LLPT whitespace closely follow the semantics of most hand-generated scanners. At each invocation the scanner first strips away whitespace and then finds a token.

5.3. Conclusions on Grammar Specification

Thus an LLPT grammar is more detailed than an ordinary one in the sense that its tokens are described, not merely mentioned. In fact, it completely specifies a language. However, the use of regular expressions to describe tokens opens the way to certain ambiguities that would not be noticed in cer-

tain systems. These ambiguities are resolved in a natural way. In addition, a language construct is introduced which allows one to define whitespace conventions.

6. Tree Construction

Since the goal of LLPT is to produce entire interpreters and translators, it is not enough simply to recognize a language. Once input has been established as being correct some steps must be taken to perform the indicated activity. This synthesis phase can be performed during parsing (with reduction-time actions), or after parsing. LLPT permits the former just as yacc does, in the form of semantic actions associated with rules. Alternately, a parse tree can be constructed to represent the program for later consultation. Semantic activity can then be deferred until the program is fully parsed. LLPT anticipates this by constructing parse trees automatically. The details of parse tree construction are discussed in this section.

6.1. Parse Trees

Parse trees are rooted trees in which each node is closely related to some specific grammar production. LLPT must do two things before automatic parse tree construction is possible. First, it must be able to determine from the grammar what nodes corresponding to a given production will look like. This information is used to define C++ classes for each production so each node can be implemented as an object instantiated from the appropriate class. Second, LLPT must construct a parser that builds a parse tree as it recognizes programs.

LLPT constructs abstract, rather than concrete, parse trees. Whereas there is an isomorphism between nodes in concrete parse trees and grammar productions, abstract parse trees are composed of nodes which correspond less closely to grammar productions. LLPT grammars carefully define the nature of the abstract tree. First, abstract trees do not contain leaves corresponding to literal terminals (keywords, operators, etc.). LLPT is able to identify literal terminals because they are enclosed in single quotes. Second, nodes corresponding to chain rules are eliminated. LLPT finds chain rules by looking for the angled brackets that are used to indicate them. Finally, lists are represented as a single parent

node with many children rather than as tall trees. Lists are identified by the square brackets followed by regular expression operators '+', '?', and '*'.

Although there are a number of convenient representations for such trees, it seems natural (and will in fact prove helpful later) to use C++, because nodes can then be instances of classes, and the user can easily add to these classes fields and methods which are associated with particular semantics.

The primary advantages of this scheme are: (1) it is possible to hide the internal representation of the tree as long as one gives the user a convenient way to access a given node's important attributes; (2) the instantiation and destruction of nodes become trivial, given the appropriately defined methods associated with these tasks; and most importantly (3) semantic actions (like type-checking and evaluation) are generally associated with grammar productions, and it will be helpful to include functions for these tasks as methods of a given class.

The following is the inheritance hierarchy for LLPT nodes, which may have three levels. First, there is a base class `node` which looks as follows:

```
class node {
    slist children; // pointers to children
    node *parent; // pointer to parent
public:
    // Identification members
    int production; // production name
    int alternation; // alternation number
    // Informational members
    int cloop; // controls implicit looping
    int numchildren() { return children.howmany(); }
    // Inquiry members
    void *getchild(int n) { return children.get(n); }
    node *getparent() { return parent; }
    // Construction members
    void setparent(node *n) { parent=n; }
    void addchild(void *n) { children.append(n); }
};
```

This class contains methods and fields common to all nodes. Ideally, the class members presented above transparent to the user; they are used by the parser as it constructs trees and by C++ code that uses implicit looping or child access. However, they are available to the user should finer control be neces-

sary.

There is also a class for each nonterminal symbol appearing in the grammar. For instance, if there is a nonterminal

```
stmt: expr ';'
      ;
```

then there will be a class `stmt`. If there is only one `stmt` alternation, then `stmt` will look as follows.

```
class stmt: public node {
public:
    stmt(node * c1,char * c2)
    {
        production=prod_stmt;
        alternation=1;
        cloop=2;
        addchild(c1);
        (*(node *)getchild(1)).setparent(this);
        addchild(c2);
    }
    stmt()
    {
        production=prod_stmt;
        alternation=1;
        cloop=2;
    }
    ~stmt()
    {
        delete getchild(1);
        delete getchild(2);
    }
};
```

Note that (so far) class `stmt` has essentially one function: to define constructors and destructors specific to that production. By and large a user will never invoke these: they are called automatically by the parser as the tree is built.

So far there are two levels to the class hierarchy: first, a node, and second, a class associated with a production. However, a production may have more than one alternation. In this case, a third level is added to the hierarchy. To illustrate this, consider these two productions:

```

stmt: expr ';'
    ;
expr: expr '+' expr
    | expr '*' expr
    | number
    ;

```

The classes defined for these will have the following outline.

```

class node {
    // all the stuff a node should have
};
class stmt: public node {
    // all the stuff a stmt should have
};
class expr: public node {
    // an empty container class
};
class expr_1: public expr {
    // all the stuff for the '+' alternation
};
class expr_2: public expr {
    // all the stuff for the '*' alternation
};
class expr_3: public expr {
    // all the stuff for the 'number' alternation
};

```

A parse tree will be built out of instances of these classes. In order to use the tree properly, it will be necessary for the user to become familiar with the scheme used here; the best way to do this is to write a small example and look at the output.

7. Semantic Actions

The features presented thus far compose a fine platform for language description, but it is still necessary to introduce a mechanism which assists the user in assigning meaning to whatever programs are recognized during parsing. LLPT's assumption is that this meaning, the semantics of a program, will be defined by the actions caused by the user either as the program is parsed or after parsing is complete. In either case, these actions will revolve around the number and kinds of productions which are recognized. In fact, the close relationship between actions and grammar rules is one of the motivations for using C++ class instances as parse tree nodes: one can implement these actions as methods of the

parse tree nodes. These are then available for invocation by the user either as the parse tree is built (with reduction-time actions, precisely as in yacc) or after the tree has been constructed.

7.1. Example

Consider the following grammar production:

```
stmt: "while" "(" expr ")" stmt
      ;
```

Suppose one wishes to assign a meaning to this production which corresponds to the expected iterative notion. Then it is sufficient to describe the process of evaluating such a statement node with C++ code. For example, we would like to associate the following evaluation method with the above production:

```
// evaluate a node corresponding to the "expr" production
void eval() {
    while ($expr->eval() != 0)
        $stmt->eval();
}
```

This fragment assumes that nodes of type `expr` have an `int eval()` method attached. It also assumes, correctly, that `$expr` and `$stmt` are recognized by LLPT and replaced with appropriate genuine C++ code that retrieves the correct children (this will be discussed later).

The syntax for this rule/method association is that any production may be followed by a dot, in which case the code between the next set of curly braces is enclosed in the class definition of that node.

For example

```
stmt: "while" "(" expr ")" stmt
      . { // '.' introduces method definitions.
          void eval() {
              while ($expr->eval() != 0)
                  $stmt->eval();
          }
      }
      ;
```

It is now possible, given a node of type `stmt`, to invoke its `eval()` method to perform the indicated task.

This could be done at reduction time as follows:

```

stmt: 'while' '(' expr ')' stmt { $$->eval(); }
    . {
        void eval {
            while ($expr->eval()!=0)
                $stmt->eval();
        }
    }
;

```

Alternatively it could be done later on, probably after the entire tree is built. Note that the reason for the "." above is to distinguish between a reduction action and a class method definition.

Two kinds of preprocessing are done to the C++ code within the method definition braces. The first involves the dollar symbol \$, which signals that the user wishes to refer to a child node corresponding to either a terminal or nonterminal on the right-hand-side of the given production. It is followed by an identifier that must match a nonterminal in the production. Thus in the above code, \$expr refers to the first child of the current node (remember that simple terminals are elided from the tree). If there is more than one child with the same name, it is possible to use a # suffix: a second expr would be referred to as \$expr#2, and so on.

Because the user may wish to access child nodes from plain C++ codes, it is necessary for the user to understand the precise translation of such expressions. The above \$expr is translated into ((expr *)getchild(1)). That is, the first child is retrieved, and because children are stored as void pointers it is necessary to cast the return value of getchild(1) into an expr pointer. It is always desirable to cast the return value of getchild() into the a pointer of the class at the lowest level of the node hierarchy, because then one has access to the most specific methods.

One problem which arises from the use of C++ involves productions with more than one alternation. Suppose we were to define an expr production for use with the above while loop:

```

expr: expr '+' expr
    . {
        int eval() { return $expr->eval() + $expr#2->eval(); }
    }
    | expr '*' expr
    . {
        int eval() { return $expr->eval() * $expr#2->eval(); }
    }
    | "[0-9]+"
    . {
        int eval() { return atoi($"[0-9]+"); }
    }
    ;
stmt: 'while' '(' expr ')' stmt { $$->eval(); }
    . {
        void eval() {
            while ($expr->eval() != 0) // !!!
                $stmt->eval();
        }
    }
    ;

```

Note that `expr` has more than one production, with a different evaluation scheme for each one. The `$expr` reference in the line marked `!!!`, however, is by itself a bug. C++ is unable to resolve this reference at compile-time, because it doesn't know that each class which inherits from node `stmt` contains an `eval()` method. This must be explained to the C++ compiler by inserting **virtual** function declarations into the `expr` node declaration. LLPT permits this as follows.

```

expr ~
{
    virtual int eval(); // function declaration
}
expr '+' expr
. {
    int eval() { return $expr->eval() + $expr#2->eval(); }
}
|
... etc ...

```

Now the expression `$expr->eval()` is meaningful, because care has been taken to inform C++ that each class associated with the `expr` alternations has an `eval()` function.

Remark: A perfect prototyping tool would insert such declarations automatically. In addition, it would extract method definitions from the class definitions and place them in a separate file, enabling one

to use the class header file without obtaining "multiple function declaration" error messages. This has not been done largely because of the difficulty involved in creating a C++ parser, and also because these are intrinsically C++ failures that might well be resolved by choosing a better object-oriented language as a base for LLPT.

A second preprocessing stage involves productions which use the extended BNF for lists. Suppose one has the following production for statement lists.

```
stmtlist: [ stmt ';' ]*  
        ;
```

then the evaluation of `stmtlist` would probably involve a straightforward iteration over each of the `stmt` children. This could be accomplished as follows:

```
stmtlist: [ stmt ';' ]*  
        . {  
            // possible C++ code...  
            [ $stmt.eval(); ]  
            // possible C++ code...  
        }  
        ;
```

Here, any C++ statements within the '[' and ']' are placed in a loop which is executed once for each occurrence of a statement. Note that the `$stmt` reference is changed into code which appropriately references each successive statement node.

8. Summary

LLPT provides mechanisms which assist in the language implementation task. It provides a way to specify the entire grammar of the language, notation to assist the user in building a parse tree of the correct shape, and a way to associate semantic actions with nodes of the parse tree.

CHAPTER IV CONCLUSION

1. Advantages

At this point, it is appropriate to step back and look at the results of this effort. First, a unified and compact approach to a large portion of syntactic specification has been developed in which many advantages of the traditional approach are retained. The solution is superior to other attempts in two ways. It can be judged more convenient than the scannerless parsing method because of its ability to handle whitespace easily and disambiguating rules. In addition, it exhibits a flexibility (with respect to whitespace and lexical environment) that is not evidenced by other scanner-implication systems.

The second issue involves parse trees. The problem of correlating abstract syntax with concrete syntax is a difficult one. The approach which specifies abstract syntax with reduction-time actions is completely flexible but opaque. The CAGT approach is reasonably flexible but obscures the relationship between concrete and abstract syntax. The idea that one should be able to specify an abstract syntax by annotating a concrete grammar seems a good one. The BNF extensions proposed here serve that purpose.

The final issue involves semantics. The fact that both denotational semantics and attribute grammars tie semantics to grammar rules suggested an object-oriented approach to semantic synthesis. LLPT parse tree nodes are instantiations of C++ classes, and the user is capable of tailoring these classes with the appropriate methods and fields.

There is no question that the three issues can be wrapped up into a neat bundle, because this has in fact been done in LLPT. It is not a production tool, but has served well in the role of experimental platform. The appendix contains a working LLPT program that defines the syntax and semantics of a simple desk calculator.

2. Drawbacks

There are a number of problems with LLPT. Most can be solved with a reasonable amount of effort. Here is a list of the most irritating flaws.

2.1. C++

In the same way that yacc is associated with C, LLPT is associated with C++. The reasons for the "upgrade" are clear: C++ is an object-oriented language, and something akin to its class mechanism is clearly called for by LLPT. However, it may be that C++ is not the best basis for LLPT. It is an exceedingly complicated language and, although there may be advantages to the experienced user, a tremendous amount of effort is required to learn even the minimal amount required of LLPT users. Only time will tell whether the tradeoff is favorable.

2.2. Interpretation

In addition, any tool which claims to assist in prototyping ought to have an interpretive mode to cut down a part of the write-compile-test cycle that is prevalent in software development. It would be nice if the semantic actions of LLPT could be specified in a language that could be interpreted; this would pave the way for a fully interpreted prototyping tool, rather than one which requires the compilation of a scanner, a parser, and whatever C++ code is included in the LLPT specification.

2.3. Conditional Compilation

In any case, a polished product would understand how to deal with incremental changes to an interpreter specification. For instance, changes to a regular expression embedded in the grammar should not necessarily cause a new parser to be built.

3. Suggestions

LLPT is not a finished product, in the sense that there are a number of things which could and should be done to improve it. Areas for further exploration are discussed below.

3.1. C++ Libraries

Although it is not at all clear that C++ provides an appropriate basis for a prototyping tool, the benefits of object-oriented programming, particularly with respect to this kind of task, are clear. Thus, regardless of which language is appropriate, an effort should be made to find the abstractions which are useful to the task of language development; once these are determined, an effort should be made to equip LLPT with a suitable library of abstract data types corresponding to these abstractions. Offhand, there are a handful of obvious choices: symbol table management, environment management, and type checking could each be facilitated by the provision of C++ classes corresponding to each abstraction. No doubt there are a host of candidates, and with a little thought such a library could be made useful not only to LLPT users but anybody who does language development in C++.

3.2. Error Messages

To date, little effort has gone into reporting errors in a clean and helpful way. No prototyping system should lack useful diagnostics. LLPT should generate helpful messages to assist the user when bugs are introduced in the grammar, the regular expressions embedded in the grammar, or the C++ code which is also embedded in the grammar.

REFERENCES

1. B. W. Kernighan and D. M. Ritchie. "The C Programming Language", Prentice-Hall, Englewood Cliffs, NJ, 1978.
2. M. E. Lesk, "Lex - A Lexical Analyzer Generator", Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.
3. S. C. Johnson, "Yacc - Yet Another Compiler-Compiler", Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.
4. D. J. Salomon and G. V. Cormack, "Scannerless NSLR(1) Parsing of Programming Languages", SIGPLAN Notices, Volume 24, Number 7, July 1989.
5. R. W. Gray, V. P. Heuring, S. P. Krane, A. M. Sloane, and W. M. Waite, "Eli: A Complete, Flexible Compiler Construction System", Software Engineering Group Report No. 89-1-1, Department of Electrical and Computing Engineering, University of Colorado, Boulder, CO, June 12, 1989.
6. U. F. Pleban and P. Lee, "An Automatically Generated, Realistic Compiler for an Imperative Programming Language", SIGPLAN Notices, Volume 23, Number 7, July 1988: 222-232.
7. J. C. H. Park, "Y+: A Yacc Preprocessor for Certain Semantic Actions", SIGPLAN Notices, Volume 23, Number 6, June 1988: 97-106.
8. A. Bahrani, "CAGT Reference Guide", Compiler Tools Group (ELI) Department of Electrical Engineering and Computer Science, University of Colorado, Boulder, CO, October 1988.
9. T. W. Reps, "Generating Language-Based Environments", ACM Doctoral Dissertation Awards, MIT Press, Cambridge, MA, 1983.
10. T. W. Reps and T. Teitelbaum, "The Synthesizer Generator: A System for Constructing Language-Based Editors", Springer-Verlag, New York, 1988.
11. C. N. Fischer and R. J. LeBlanc, Jr., "Crafting A Compiler", Benjamin/Cummings, Menlo Park, CA, 1988.

12. U. W. Pleban, "Compiler Prototyping Using Formal Semantics", SIGPLAN Notices, Vol. 19, Number 6, June 1984: 94-105.
13. L. Paulson, "A Semantics-Directed Compiler Generator", 9th ACM POPL Conference, Albuquerque, NM, January 1982: 224-239.
14. M. Raskovsky, "Denotational Semantics as a specification of code Generators", Proceedings ACM SIGPLAN '82 Conference on Compiler Construction, SIGPLAN Notices 17, number 6, June 1982: 230-244.
15. J. McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine", Communications of the ACM, Volume 3, Number 4, 1960: 184-195.
16. B. Stroustrup, "The C++ Programming Language", Addison-Wesley, Reading, MA, 1986.

Appendix A: A desk calculator

```

/*
 * What follows is LLPT code for a small desk calculator. It
 * supports integer arithmetic, variables and functions with
 * single-letter names, and various looping and conditional
 * constructs.
 */

%{
    extern double  atof(const char *);
    double  memory[26];          /* variable storage */
    node *functions[26];        /* function storage */
}%

%start    program

/* The usual notions of precedence and associativity
 */
%right '='
%left '|'
%left '&&'
%left '==' '!='
%left '<' '<=' '>' '>='
%left '+' '-'
%left '*' '/'
%left "UMINUS"

/* Whitespace is zero or more blanks, tabs, newlines, or C++
 * comments.
 */
%whitespace    "([\ \t\n]|\(\s\|.*)*"

%%

expr ~
{
    // All expr's have an eval() method
    virtual double eval() { cout << "Undefined; sorry\n"; return 0; };
}
|[a-z]" '=' expr
. {
    // assignment. Note overloaded '$'.
    double eval() { return memory[*$"[a-z]"-'a']=$expr->eval(); }
}
|[a-z]" '(' actuals ')'
. {
    // methods can be declared now and defined later.
    double eval();
}
}

```

```

| "[a-z]"
.{
    double eval() { return memory[*$"[a-z]"-'a']; }
}
| expr '>=' expr
.{
    // Note the disambiguating suffix on $expr#2.
    double eval() { return $expr->eval() >= $expr#2->eval(); }
}
| expr '<=' expr
.{
    double eval() { return $expr->eval() <= $expr#2->eval(); }
}
| expr '==' expr
.{
    double eval() { return $expr->eval() == $expr#2->eval(); }
}
| expr '!=' expr
.{
    double eval() { return $expr->eval() != $expr#2->eval(); }
}
| expr '<' expr
.{
    double eval() { return $expr->eval() < $expr#2->eval(); }
}
| expr '>' expr
.{
    double eval() { return $expr->eval() > $expr#2->eval(); }
}
| expr '-' expr
.{
    double eval() { return $expr->eval() - $expr#2->eval(); }
}
| expr '+' expr
.{
    double eval() { return $expr->eval() + $expr#2->eval(); }
}
| expr '/' expr
.{
    double eval() { return $expr->eval() / $expr#2->eval(); }
}
| expr '*' expr
.{
    double eval() { return $expr->eval() * $expr#2->eval(); }
}
| '-' expr %prec "UMINUS"
.{
    double eval() { return -$expr->eval(); }
}
| "[0-9]+"
.{

```

```

        double eval() { return atof($"[0-9]+"); }
    }
    | < '(' expr ')' > /* Chain rule: does not require an eval */
    ;

stmt ~
{
    // All stmts have an eval as well. The value of a stmt is
    // the value of the last expression evaluated.
    virtual double eval() {
        cout << "unimplemented evaluation scheme for stmt\n";
        return 0;
    }
}

'{' stmtlist '}'
. {
    double eval();
}
| expr ';'
. {
    double eval() { return $expr->eval(); }
}
| 'print' expr ';'
. {
    double eval() {
        double   retval;
        retval=$expr->eval();
        cout << "Result:" << retval << "\n";
        return retval;
    }
}
|
'while' '(' expr ')' stmt
. {
    double eval();
}
| 'for' '(' expr ';' expr ';' expr ')' stmt
. {
    double eval();
}
| 'if' '(' expr ')' stmt
. {
    double eval() {
        double retval=0;
        if ($expr->eval())
            retval=$stmt->eval();
        return retval;
    }
}
| 'if' '(' expr ')' stmt 'else' stmt
. {
    double eval() {

```

```

        if ($expr->eval())
            return $stmt->eval();
        else return $stmt#2->eval();
    }
}
| 'func' "[a-z]" '(' formals ')' stmt
. {
    // evaluating a function definition just stores
    // the function body and argument list
    double eval() {
        int c;
        c="$[a-z]"-'a';
        functions[c]=this;
        return 0;
    }
}
;

/* Function parameters */
formals :
    "[a-z]"
    | formals ',' "[a-z]"
;

/* Function argument list */
actuals: expr
    | actuals ',' expr
;

stmtlist: [ stmt ]*
    . {
        double eval();
    }
;

/* Note the reduction-time evaluation */
program: [ stmt { $stmt->eval(); }]*
%%
main()
{
    extern int yyparse();
    extern node *yytree;

    yyparse();
}

double stmt_1::eval() { return ((stmtlist *)getchild(1))->eval(); }

double stmtlist::eval() {
    double retval;

```



```

    for (int yyloop=0; yyloop<numchildren(); yyloop+=cloop)
        retval=((stmt *) getchild(1+yyloop))->eval();
    return retval;
}

double stackvals(stmt *body, formals *f, actuals *a)
{
    // let C++ manage recursive stacks
    char c;
    double    retval, save;

    c=(char *)f->getchild(f->alternation);
    save=memory[c-'a'];
    memory[c-'a']=((expr *)a->getchild(a->alternation))->eval();
    f=(formals *)((f->alternation==2)?f->getchild(1):NULL);
    a=(actuals *)((a->alternation==2)?a->getchild(1):NULL);
    retval=(f && a)?stackvals(body, f, a):body->eval();
    memory[c-'a']=save;
    return retval;
}

double expr_2::eval()
{
    node *func;
    stmt *body;
    formals *f;
    actuals *a;
    char c;

    c=(char *)getchild(1);
    func=functions[c-'a'];
    body=(stmt *)func->getchild(3);
    if (!func) {
        cerr << "Undefined function " << c << "\n";
        return 0;
    }
    f=(formals *)func->getchild(2);           // get formals
    a=(actuals *)func->getchild(2);         // get actuals
    if (f && a)
        return stackvals(body, f, a);
    else return body->eval();
}

double stmt_4::eval()
{
    double retval=0;
    while (((expr *) getchild(1))->eval())
        retval=((stmt *) getchild(2))->eval();
    return retval;
}

```

```
double stmt_5::eval()
{
    double retval=0;
    ((expr *) getchild(1))->eval();
    while (((expr *) getchild(2))->eval()) {
        retval=((expr *) getchild(4))->eval();
        ((expr *) getchild(3))->eval();
    }
    return retval;
}
```

Appendix B: Source listings

Because LLPT is based on lex and yacc, which are copywrited programs, it is not possible to produce the entire text of LLPT. Thus this listing includes only the output of the Unix diff command invoked on the myacc/yacc and mlex/lex source directories. The truly interested reader should be able to piece the source together with this listing and the original lex and yacc source files.

First, the difference between the parser and yacc.

```

    Only in myacc: lexer.awk
BEGIN {
    FS=""
    while (getline > 0) {
        if ($0=="%%" && $0!~":") break
        if ($0=="%tokprec" && $0!~":")
            TP++
        else TOKPREC[$1,$2]=TP
    }
    while (getline > 0) {
        if ($0=="%%" && $0!~":") break
        sub(/:$/, "", $2)
        WSPACE[$1]=$2
    }
    while (getline > 0)
        print $0 | "sort +1 -2 | sort > lexer2.tmp"
    close ("sort +1 -2 | sort > lexer2.tmp")
    MAXSTATE=0
    PREVSTATE=""
    exit
}
# TOKPREC[expr:env SUBSEP [10]]=prenum
# BASESTATES[statenum] exists and is -1 if it is a "core" state
# SAMESTATES[statenum] otherwise; set to number of the equivalent core state
# TOKVAL[expr:env]=tokenvalue >256
# TOKENV[expr:env]=env
# TOKSTR[expr:env]=expr
END {
while (getline < "lexer2.tmp" > 0) { # < "lexer2.tmp" > 0) {
    if ($1>MAXSTATE) MAXSTATE=$1;
    if (PREVSTATE=="") PREVSTATE=$1
    if ($1!=PREVSTATE) {
        if (!(STERMS in DIFFSTATES)) {
            DIFFSTATES[STERMS]=PREVSTATE
            BASESTATES[PREVSTATE]=-1;
        } else SAMESTATES[PREVSTATE]=DIFFSTATES[STERMS]
        STERMS=""
        PREVSTATE=$1
    }
}
}

```

```

}
base=3
if (NF>6)
    for (i=3; i+4<=NF; i++) {
        $2=$2 "" $i
        base=i+1
    }
num=split($2,stuff,":")
if (num>2) {
    for (i=2; i+1<=num; i++)
        stuff[1]=stuff[1] ":" stuff[i]
    stuff[2]=stuff[num]
}
TOKENV[$2]=stuff[2]
TOKVAL[$2]=$NF-3
TOKSTR[$2]=stuff[1]
TOKLIT[$2]=$NF
STERMS=STERMS SUBSEP $2
}
if (!(STERMS in DIFFSTATES)) {
    DIFFSTATES[STERMS]=PREVSTATE
    BASESTATES[PREVSTATE]=-1;
} else SAMESTATES[PREVSTATE]=DIFFSTATES[STERMS]
output()
#system("rm -f lexer2.tmp")
}

func output(    start, i, sts, num)
{
    print "%{"
    print "void yyterror(short);"
    print "#include <stdlib.h>"
    print "#include <strings.h>"
    print "#include <stream.h>"
    print "#include
    print "extern short yyystate;0hort yyyswitch;48}"
    printf "%%start dummy "
    for (i in DIFFSTATES)
        printf " S%d", DIFFSTATES[i]
    printf "48%%%0{0
    print "switch(yyystate) {"
    for (i=0; i<=MAXSTATE; i++) {
        if (i in BASESTATES) {
printf("case %d: BEGIN S%d; yyyswitch=%d; break;0, i, i, i)
        } else if (i in SAMESTATES) {
printf("case %d: BEGIN S%d; yyyswitch=%d; break;0, i, SAMESTATES[i], SAMESTATES[i])
        }
    }
    print "}"
    for (wspaces in WSPACE) {
        if (WSPACE[wspaces]) {

```

```

        printf "<dummy"
        for (stokens in DIFFSTATES) {
            num=split(stokens, sts, SUBSEP)
            x=TOKENV[sts[2]]
            if (x==wspaces)
                printf(",S%d",DIFFSTATES[stokens]);
        }
        printf(">%s;0, WSPACE[wspaces])
    }
}
for (stokens in DIFFSTATES) {
    num=split(stokens, sts, SUBSEP)
    x=TOKENV[sts[2]]
    for (i=3; i<=num; i++) {
        if (x!=TOKENV[sts[i]]) {
print x, TOKENV[sts[i]], sts[i], i, num
            error("Error in state " DIFFSTATES[stokens] ": which whitespace to strip?")
        }
    }
    for (j=2; j<=num; j++)
        soutput(sts[j], DIFFSTATES[stokens])
}
print ".|\n{ yylsterror(yyyswitch); exit(-1); }"
print "%%"
print "yywrap()return -1;0"
yylsterror()
}

func soutput(t, s, i, ntstr,c)
{
    if (TOKLIT[t]==0) {
        ntstr=TOKSTR[t]
        gsub("
printf("<S%d>
        } else printf("<S%d>%s{yylval.strptr=new char[1+strlen(yytext)];0strcpy(yyval.strptr, yytext);0return %d;};0, s, TOI
    }

func stroutput(t, s,i, ntstr,c)
{
    if (TOKLIT[t]==0) {
        ntstr=TOKSTR[t]
        gsub("
        printf("
    } else printf("
}

func yynsterror( start, i)
{
    print "void yynsterror(short i)0"
    printf("fprintf(stderr,
    for (i=0; i<MAXSTATE; i++) {

```

```

        if (i in BASESTATES) {
            if (start)
                printf "} else"
            start=1
            printf "if"
            printf("(i==%d) {0, i)
            errinfo(i)
        }
    }
    print "}"
    printf("fprintf(stderr,
    print "}")
}

func errinfo(i)
{
    for (stokens in DIFFSTATES) {
        if (DIFFSTATES[stokens]!=i) continue;
        num=split(stokens, sts, SUBSEP)
        x=TOKENENV[sts[2]]
        for (j=2; j<=num; j++) {
            printf("fprintf(stderr,
            stroutput(sts[j], DIFFSTATES[stokens])
            printf(");0);
        }
    }
}

func error(s)
{
    print s | "cat 1>&2"
    exit(1)
}

Only in myacc: classes.awk
BEGIN {
    inf=ARGV[1]
    while (getline < inf > 0) {
        if ($0~/startdecl/) {
            while ((getline < inf > 0) && $0!~/enddecl/)
                ;
            getline < inf
        }
        if ($0~/starteval/) {
            while ((getline < inf > 0) && $0!~/endeval/)
                ;
            getline < inf
        }
        maxes[$1]++
    }
    close(inf)
}

```

```

/startdecl/ {
    decl=""
    while ((getline>0) && $0!~/enddecl/)
        decl=decl "0 $0
    getline
}
/starteval/ {
    eval=""
    while ((getline>0) && $0!~/endeval/)
        eval=eval "0 $0
    getline
}

```

```

$0 != "" {
    if (maxes[$1]>1 && $2==1) {
        print "Olass " $1 ":public node {"
        print "public:"
        print decl
        print "};"
    }
    if (maxes[$1]>1)
        cn=$1 " " $2
    else cn=$1
    if (maxes[$1]>1)
        print "Olass " cn ": public " $1 " {"
    else print "Olass " cn ": public node {"
    print "public:"
    printf "" cn "("
    for (i=3; i<NF; i++)
        printf xval($i) " c" i-2 " ,"
    if (NF>=3)
        printf xval($i) " c" i-2
    print ")0{"
    print "production=prod_ " $1 " ;"
    print "alternation=" $2 " ;"
    print "cloop=" NF-2 " ;"
    for (i=3; i<NF; i++) {
        print "addchild(" mn(i-2) " );"
        if ($i=="nptr")
            print "(*(node *)getchild(" i-2 ")).setparent(this);"
    }
    if (i==NF)
        print "addchild(" mn(i-2) " );"
    if ($i=="nptr")
        print "(*(node *)getchild(" i-2 ")).setparent(this);"
    print "}"
    # for empty init
    if (NF>2) {
        print "" cn "00{"
    }
}

```

```

        print "production=prod " $1 ";"
        print "alternation=" $2 ";"
        print "cloop=" NF-2 ";"
        print "}"
    }
    print "" cn "00{"
    for (i=3; i<NF; i++)
        print "delete getchild(" i-2 ");"
    print "delete getchild(" i-2 ");"
    print "}"
    print eval
    print ";;"
    if (maxes[$1]==1) {
        print "0lass " $1 " _1: public " $1 "{"
        print "public:"
        print "//empty container class"
    printf "" $1 " _1("
    for (i=3; i<NF; i++)
        printf xval($i) " c" i-2 ";,"
    if (NF>=3)
        printf xval($i) " c" i-2
    printf ") : ("
    for (i=3; i<NF; i++)
        printf " c" i-2 ";,"
    if (NF>=3)
        printf " c" i-2
    print ") {}"
        print "};;"
    }
}

```

```

func mn(i)
{
    if ($(i+2)=="nptr") return "c" i
    return "c" i
}

```

```

func xval(x)
{
    if (x=="nptr") return "node *"
    return "char *"
}

```

Only in yacc: y5.c

```
#include <stdio.h>
```

```
#include "dextern"
```

```

extern FILE*fclasses, *faction;
extern int *mem;
extern int *prdptr[NPROD];
extern char *typeset[NTYPES];
extern char *symnam();

```



```

closetmp(np, end, special)
int np;
int end;
int special;
{
    int offset, i, j, tmp;

    offset = mem-prdptr[np]-1;
    offset -= end;
    fprintf(fclasses, "%s ", symnam(*prdptr[np]));
    for (j=0,i=1; i<=np; i++)
        if (*prdptr[np]==*prdptr[i]) j++;
    fprintf(fclasses, "%d", j);
    for (i=1; i<=offset; i++) {
        if (absym(prdptr[np][i]))
            continue;
        tmp=fdtype(prdptr[np][i]);
        fprintf(fclasses, " %s", typeset[tmp]);
    }
    fprintf(fclasses, "0");
}

buildtree(np, end, special)
int np;
int end;
int special;
{
    int offset, i, j, k, m, tmp;
    extern int chainrule[NPROD];

    offset = mem-prdptr[np]-1;
    offset -= end;
    fprintf(faction, "0* build tree */0);
    if (chainrule[np]) {
        fprintf(faction, "0* chain rule */0);
        for (i=1; i<=offset; i++) {
            if (absym(prdptr[np][i])) continue;
            if (prdptr[np][i]>=NTBASE)
                if (prdptr[np][i]!=*prdptr[np])
                    error("chain rule will foul up C++");
            tmp=fdtype(prdptr[np][i]);
            fprintf(faction, "yyval.nptr=yyvspvt[%d].%s",
                i-offset, typeset[tmp]);
            i++;
            break;
        }
        if (i>offset)
            error("chain rule without significant elements");
        for (; i<=offset; i++) {
            if (absym(prdptr[np][i])) continue;
            error("chain rule with more than 1 significant elt");
        }
    }
}

```

```

    }
    return;
}
for (m=0, i=1; i<=offset; i++) {
    if (absym(prdptr[np][i])) continue;
    m++;
}
for (j=0, i=1; i<=np; i++)
    if (*prdptr[np]==*prdptr[i]) j++;
if (special) {
    if (offset) {
        for (k=2; k<=offset; k++) {
            if (absym(prdptr[np][k]))
                continue;
            tmp=fdtype(prdptr[np][k]);
            fprintf(faction, "(*(yypvt[%d].nptr)).addchild(",
                1-offset);
            fprintf(faction, "yypvt[%d].%s",
                k-offset, typeset[tmp]);
            fprintf(faction, ");");
        }
        } else fprintf(faction, "yyval.nptr=new %s_%d(",
            symnam(*prdptr[np],j-1);
    } else fprintf(faction, "yyval.nptr=new %s_%d(",
        symnam(*prdptr[np],j);
    for (i=1; i<=offset; i++) {
        if (absym(prdptr[np][i]))
            continue;
        tmp=fdtype(prdptr[np][i]);
        if (!special || !offset) {
            fprintf(faction, "yypvt[%d].%s", i-offset, typeset[tmp]);
            /* if (i<offset) fprintf(faction, ","); */
            if (--m) fprintf(faction, ",");
        }
    }
    if (!special || !offset)
        fprintf(faction, ");");
}
}
Only in myacc: node.h
class node {
    slist children; // pointers to children
    node *parent; // pointer to parent
public:
    int production; // production name
    int alternation; // alternation number
    int cloop; // how many rhs were there?
    node() {
        parent=0;
        production=0;
        alternation=0;
    }
}

```

```

void *getchild(int n) { return children.get(n); }
int  numchildren() { return children.howmany(); }
node *getparent() { return parent; }
void setparent(node *n) { parent=n; }
void addchild(void *n) { children.append(n); }
};

```

Only in myacc: slist.C

```

#include <stdlib.h>
#include <stream.h>
#include "slist.h"

```

```

int  slist::append(void *a)
{
    num++;
    if (last) {
        last->next = new slink(a, 0);
        last = last->next;
    }
    else first = last = new slink(a, 0);
    return 0;
}

```

```

void *slist::get(int n)
{
    void default_error(char *);

    if (n>num) default_error("not that many elements");
    if (n<0) default_error("negative get");
    slink *tmp=first;
    for (int i=1; i<n; i++)
        tmp=tmp->next;
    return tmp->e;
}

```

```

void slist::clear()
{
    while (first) {
        slink *tmp=first->next;
        delete first;
        first = tmp;
    }
    first = last = 0;
    num = 0;
}

```

```

void default_error(char *s)
{
    cerr << s << "\n";
    exit(1);
}

```

Only in myacc: slist.h

```

class slink {
friend class slist;
    slink *next;
    void *e;
    slink(void *a, slink *p) { e=a; next=p; }
};

class slist {
    slink* last;
    slink* first;
    int    num;
public:
    int    append(void *a); // add at tail of list
    void *get(int n);      // return and remove head of list
    void clear();          // remove all links
    int    howmany() { return num; } // how many links?

    slist()    { first=last=0; num=0;}
    slist(void *a)    { first=last=new slink(a, 0); num=1;}
    ~slist()    { clear(); }
};

```

diff yaccpar.C yaccpar

4a5

> /* #define YYDEBUG */

10d10

<

19a20,24

> short yyystate;

> extern int yylex(), yyylex();

> extern void yyerror(char *);

> extern int yyparse();

> node *yytree;

21c26,27

< yyparse() {

> int yyparse()

> {

30c36

< yyystate = 0;

> yyystate = yyystate = 0;

42c48,50

< if(++yyps> &yyys[YYMAXDEPTH]) { yyerror("yacc stack overflow"); return(1); }

> if(++yyps> &yyys[YYMAXDEPTH]) {

> yyerror("yacc stack overflow"); return(1);

> }

53c61

< if(yychar<0) if((yychar=yylex())<0) yychar=0;

> if(yychar<0) if((yychar=yyylex())<0) yychar=0;

```

59c67
<     yystate = yyn;
---
>     yyystate = yystate = yyn;
68c76
<     if( yychar<0 ) if( (yychar=yylex0)<0 ) yychar = 0;
---
>     if( yychar<0 ) if( (yychar=yyylex0)<0 ) yychar = 0;
76c84,86
<     if( (yyn = yyxi[1]) < 0 ) return(0); /* accept */
---
>     if( (yyn = yyxi[1]) < 0 ) {
>         yytree = yyval.nptr;
>         return(0); /* accept */
77a88
>     }
86a98
>         goto yyerrlab; /* just so used */
100c112
<         yystate = yyact[yyn]; /* simulate a shift of "error" */
---
>         yyystate = yystate = yyact[yyn]; /* simulate a shift of "error" */
147c159
<     if( yyj>=YYLAST || yychk[ yystate = yyact[yyj] ] != -yyn ) yystate = yyact[yyppo[yyn]];
---
>     if( yyj>=YYLAST || yychk[ yyystate = yystate = yyact[yyj] ] != -yyn ) yyystate = yystate = yyact[yyppo[yyn]
153a166,188
>
> int yyylex0
> {
>     int i;
>     extern char *yytext;
>
> #ifdef YYDEBUG
>     if (yydebug)
>         fprintf(stderr, "LEX: (state==%d)0, yyystate);
> #endif
>     i=yylex0);
> #ifdef YYDEBUG
>     if (yydebug)
>         fprintf(stderr, "LEXRET: %d (%s)0, i, yytext);
> #endif
>     return i;
> }
>
> void yyerror(char *s)
> {
>     cerr << "There was an error: " << s << "0;
>     exit(1);
> }
diff myacc/Makefile oyacc/Makefile

```

```

1a2
> # @(#)Makefile 4.2 (Berkeley) 83/02/11
3,5c4
< # @(#)Makefile 1.1 86/09/25 SMI; from UCB 4.2 83/02/11
< #
< DESTDIR=/home/jcr/src/l1pt/
---
> DESTDIR=
8c7
< y1.c y2.c y3.c y4.c y5.c ---
> y1.c y2.c y3.c y4.c 11c10
< all: myacc
---
> all: yacc
13,17c12,17
< myacc: y1.o y2.o y3.o y4.o y5.o
< $(CC) -o myacc y?.o
< install -s myacc $(DESTDIR)/bin
< install -c yaccpar.C $(DESTDIR)/lib
< y1.o y2.o y3.o y4.o y5.o: dextern files
---
> yacc: y1.o y2.o y3.o y4.o
> $(CC) -o yacc y?.o
> y1.o y2.o y3.o y4.o: dextern files
> install:
> install -s yacc $(DESTDIR)/usr/bin
> install -c yaccpar $(DESTDIR)/usr/lib
19c19
< -rm -f *.o myacc
---
> -rm -f *.o yacc
diff myacc/dextern oyacc/dextern
243c243
< # define OFILE "y.tab.C"
---
> # define OFILE "y.tab.c"
250,266d249
< # endif
<
< /* lexer output file name */
<
< # ifndef FILELTM
< # define FILELTM "lexer.tmp"
< # endif
< /* lexer output file name */
<
< # ifndef FILEL
< # define FILEL "y.lexer.l"
< # endif
<
< /* output file for C++ class defs */

```

```

<
< # ifndef FILECPP
< # define FILECPP "classes.tmp"
diff myacc/files oyacc/files
14,19c14
< # define PARSEER "/home/jcr/src/llpt/myacc/yaccpar.C"
< /* how to build a C++ classes from a specification file */
< # define CCMD "awk -f /home/jcr/src/llpt/myacc/classes.awk classes.tmp >> y.classes.h"
< /* how to build a lex program from a specification file */
< # define LCMD "awk -f /home/jcr/src/llpt/myacc/lexer.awk lexer.tmp > y.lexer.l"
< /* basic size of the Yacc implementation */
---
> # define PARSEER "/usr/lib/yaccpar"
21,23c16,17
< #define INCFEIL1 "#include
< #define INCFEIL2 "#include
< #define HUGE
---
> /* basic size of the Yacc implementation */
> # define HUGE
diff myacc/y1.c oyacc/y1.c
40,43d39
< int indebug = 1;
< int cldebug = 0; /* debugging flag for closure */
< int gsdebug = 1;
< int pidebug = 0; /* debugging flag for putitem */
53d48
< extern int zapflag;
58,67d52
< if (indebug && foutput != NULL) {
<     int i, j;
<     for (i=0; prdptr[i]; i++) {
<         for (j=0; prdptr[i][j]>0; j++) {
<             fprintf(foutput, "%d ", prdptr[i][j]);
<         }
<         fprintf(foutput, "%d ", prdptr[i][j]);
<         fprintf(foutput, "0");
<     }
< }
79,85d63
< system(LCMD); /* build a lexer */
< system(CCMD); /* build a lexer */
< if (zapflag) {
<     ZAPFILE("classes.tmp");
<     ZAPFILE("lexer.tmp");
<     ZAPFILE("y.output");
< }
173c151
< if( *cp == ' ' || *cp == '' ) ++cp;
---
> if( *cp == ' ' ) ++cp;

```

```

177,194d154
< char sympref(i){
<   char *cp;
<
<   cp = (i>=NTBASE) ? nontrst[i-NTBASE].name : tokset[i].name ;
<   if( *cp == ' ' ) return '1';
<   if( *cp == '' ) return '0';
<   return ' ';
< }
<
< int absym(i) {
<   char *cp;
<   cp = (i>=NTBASE) ? nontrst[i-NTBASE].name : tokset[i].name ;
<   if (cp[0]==' ' || (cp[0]=='$' && cp[1]=='$'))
<       return 1;
<   return 0;
< }
<
<
304a265
> int indebug = 0;
415a377
> int pidebug = 0; /* debugging flag for putitem */
495a458
> int gsdebug = 0;
564a528
> int cldebug = 0; /* debugging flag for closure */
diff myacc/y2.c oyacc/y2.c
20,21d19
< # define WHITESPACE 271
< #define TOKPREC    272
29,30d26
< char wtag[NAMESIZE]; /* current whitespace tag */
< char swtag[NAMESIZE]; /* save current whitespace tag */
42,44d37
< int nwhtes; /* number of whitespaces defined */
< char * whiteset[NTYPES]; /* pointers to whitespace tags */
< char * whitespaces[NTYPES]; /* pointers to whitespace exprs */
51d43
< int tokprec[NTERMS]; /* store lexical precedence */
58,59d49
< /* rewind variable */
< long actplace, ftell();
64d53
< FILE * fclasses; /* file for C++ class defs */
70d58
< FILE * loutput; /* y.lexer.l file */
79,81d66
< int extbnf[NPROD] ; /* is this production +, ?, or *? */
< int chainrule[NPROD] ; /* '<' '>' rule? */
< int zapflag=1; /* zap temp files, or not? */

```



```

82a68
>
87a74
>
91,98d77
< fdefine = fopen( FILED, "w" );
< if( fdefine == NULL ) error( "cannot open header file" );
< fclasses = fopen( FILECPP, "w" );
< if( fclasses == NULL ) error( "cannot open y.classes.h file" );
< loutput = fopen( FILELTM, "w" );
< if( loutput == NULL ) error( "cannot open lexer file" );
< foutput = fopen(FILEU, "w" );
< if( foutput == NULL ) error( "cannot open y.output" );
104c83,84
< zapflag=0;
---
> foutput = fopen(FILEU, "w" );
> if( foutput == NULL ) error( "cannot open y.output" );
108c88
< fprintf(stderr, "-d option now default in yacc0);
---
> fdefine = fopen( FILED, "w" );
129,133d108
< fprintf( ftable, "#include <stdlib.h>0 );
< fprintf( ftable, "#include <stream.h>0 );
< fprintf( ftable, INCFIL1 );
< fprintf( ftable, INCFIL2 );
< fprintf( ftable, "#include
152,153d126
< typeset[++ntypes] = cstash( "strptr" );
< typeset[++ntypes] = cstash( "nptr" );
216,269d188
< case WHITESPACE:
< c = getc(finput);
< if ( c != '<' ) ungetc(c, finput);
< if ( c == '<' ) {
< int j;
< for (j=0; (c = getc(finput)) != '0' &&
< c!=' ' && c!=' ' && c!=EOF;
< j++)
< tokname[j]=c;
< if (tokname[j-1]!='>')
< error("Unterminated whitespace tag");
< tokname[j-1]=' ';
< } else tokname[0]=' ';
< if (nwhites>=NTYPES)
< error("Too many kinds of whitespace");
< for (j=0; j<nwhites; j++)
< if (!strcmp(tokname, whiteset[j]))
< error("Multiple declaration of whitespace");
< whiteset[nwhites]=cstash(tokname);

```

```

<
<     t = gettok();
<     if (!(t == IDENTIFIER &&
<         (tokname[0] == ' ' || tokname[0] == '')))
<         error("invalid whitespace definition");
<     whitespaces[nwhites]=cstash(tokname+1);
<     t = gettok();
<     ++nwhites;
<     continue;
<
< case TOKPREC:
<     fprintf(loutput, "%s\tokprec0);
<     t = gettok();
<     for(;;) {
<         switch( t ){
<         case ',':
<             t = gettok();
<             continue;
<         case ';':
<             break;
<         case IDENTIFIER:
<             if (tokname[0] != ' ' && tokname[0] != '')
<                 error("No symbolic terminals");
<             j = chfind(0,tokname);
<             fprintf(loutput, "%s\t%c0, tokname+1,
<                 (*tokname == '')?'0':'1');
<             t=gettok();
<             continue;
<
<         }
<
<         break;
<     }
<     continue;
<
297,298d215
<         if (tokname[0] != ' ' && tokname[0] != '')
<             error("No symbolic terminals");
338d254
<     fprintf( ftable, "#include
352,353c268
<     fprintf( fdefine, "class node;0);
<     fprintf( fdefine, "typedef union { char *strptr; node *nptr; } YYSTYPE;0 );
---
>     if( lntypes ) fprintf( ftable, "#ifndef YYSTYPE0define YYSTYPE int0endif0 );
355d269
<     fprintf( fdefine, "extern YYSTYPE yylval;0 );
392,398d305
<     if (t=='[') {
<         extbnf[nprod]=1;
<         t = gettok();

```

```

<     } else if (t=='<') {
<         chainrule[nprod]=1;
<         t=gettok();
<     }
400,401d306
<         if (*wtag && (*tokname==' ' || *tokname==''))
<             strcat(tokname, wtag); /* add on rule's default wspace */
420,421c325,326
<         actplace=ftell(finput);
<         cpyact((int)(mem-prdptr[nprod]-1), 0, 0);
---
>         cpyact( mem-prdptr[nprod]-1 );
>         fprintf( faction, " break;" );
423,424d327
<             fprintf( faction, "yyval.nptr=(node *) NULL;0);
<             fprintf( faction, " break;" );
448,452d350
<             /* update the ext. bnf info */
<
<             extbnf[nprod+1] = extbnf[nprod];
<             extbnf[nprod] = 0;
<
462,467c360,361
<         } else {
<             /* Last action - better do some tree
<             building here
<             */
<             buildtree(nprod, 0, 0);
<             fprintf( faction, " break;" );
---
>         }
>
469,492d362
<     }
<     if (extbnf[nprod]) {
<         if ( t != ']' )
<             error("Expecting the end of BNF brackets");
<         switch(t=gettok()) {
<             case '+': extbnf[nprod]='+'; t = gettok(); break;
<             case '?': extbnf[nprod]='?'; t = gettok(); break;
<             case '*': extbnf[nprod]='*'; t = gettok(); break;
<             default: extbnf[nprod]='+'; break;
<         }
<     }
<     else if (chainrule[nprod]) {
<         if ( t != '>' )
<             error("Expected a closing chain rule '>'");
<         t=gettok();
<     }
<     if ( t=='.' ) {
<         /* class members */

```

```

<         if (chainrule[nprod])
<             error("Attempt to attach method to chain rule");
<         cpyact((int)(mem-prdptr[nprod]-1), 1, 0);
<         t=gettok();
<     } else fprintf(fclasses, "starteval0ndeval0);
<     *mem++ = -nprod;
494,499d363
<     if (!(levprd[nprod]&ACTFLAG)) {
<         fprintf( faction, "Oase %d:", nprod );
<         buildtree(nprod,1,0);
<         closetmp(nprod,1,0);
<         fprintf( faction, " break;");
<     } else closetmp(nprod, 1, 0);
501,503d364
< if (extbnf[nprod]) {
<     int i, base;
<     base=nprod;
505,509d365
<     if (extbnf[base]=='*' || extbnf[base]=='?') {
<         if( ++nprod >= NPROD ) error( "more than %d rules", NPROD );
<         prdptr[nprod] = mem;
<         levprd[nprod]=0;
<         *mem++=prdptr[base][0];
511,531c367
<         fprintf( faction, "Oase %d:", nprod );
<         buildtree(nprod,1,extbnf[base]);
<         fprintf( faction, " break;");
<         levprd[nprod] |= ACTFLAG;
<     }
<     if (extbnf[base]=='*' || extbnf[base]=='+') {
<         if( ++nprod >= NPROD ) error( "more than %d rules", NPROD );
<         prdptr[nprod] = mem;
<         levprd[nprod]=levprd[base];
<         *mem++=prdptr[base][0];
<         for (i=0; prdptr[base][i]>=0; i++)
<             *mem++ = prdptr[base][i];
<         *mem++ = -nprod;
<         fprintf( faction, "Oase %d:", nprod );
<         buildtree(nprod,1,extbnf[base]);
<         if (levprd[nprod]&ACTFLAG)
<             cpyact((int)(mem-prdptr[nprod]-1), 0, actplace);
<         fprintf( faction, " break;");
<         levprd[nprod] |= ACTFLAG;
<     }
< }
< }
---
>
554d389
<     fprintf(loutput, "%%%%0);
557c392
<     /* fprintf( ftable, "0 line %d

```

```

---
>          fprintf( ftable, "0 line %d
561d395
< defout();
564,566c398
< finact()
< {
<   int   i, j;
---
> finact(){
570,587d401
<   fclose(fclasses);
<   fclasses=fopen("y.classes.h", "w");
<   fprintf(fclasses, "enum prodname {");
<   /* how many are there? (need to know for commas) */
<   for (j=0, i=1; i<=nnonter; i++) {
<       if (nontrst[i].name[0]!='$')
<           j++;
<   }
<   /* print them out */
<   for (i=1; i<=nnonter; i++) {
<       if (nontrst[i].name[0]!='$')
<           fprintf(fclasses, "0prod_%s", nontrst[i].name);
<       if (--j>=1)
<           fprintf(fclasses, ",0);
<       else fprintf(fclasses, "0);
<   }
<   fprintf(fclasses, "0;0);
<   fclose(fclasses);
591c405
< }
---
>   }
610,611c424
<   /* Don't care about this any more
<   if( s[0]==' ' && s[2]==' ' )
---
>   if( s[0]==' ' && s[2]==' ' ) /* single character literal */
613,615c426,429
<   else if ( s[0]==' ' && s[1]=='\ ' ) {
<       if( s[3] == ' ' ) {
<           switch ( s[2] ) {
---
>   else if ( s[0]==' ' && s[1]=='\ ' ) { /* escape sequence */
>       if( s[3] == ' ' ){ /* single character escape sequence */
>           switch ( s[2] ){
>               /* character which is escaped */
624a439
>           }
626,630c441,443
<       }

```

```

<     else if( s[2] <= '7' && s[2]>='0' &&
<         s[3] <='7' && s[3]>='0' &&
<         s[4] <= '7' && s[4]>='0' &&
<         s[5]==' ') {
---
>     else if( s[2] <= '7' && s[2]>='0' ){ /* On sequence */
>         if( s[3]<'0' || s[3] > '7' || s[4]<'0' ||
>             s[4]>'7' || s[5] != ' ') error("illegal \nnn construction" );
633,636c446,450
<     }
< }
< */
< val = extval++;
---
>     }
> }
> else {
>     val = extval++;
> }
647a462
>
649,658c464,469
<     if( *cp == ' ' || *cp == '' ) {
<         if (tokset[i].value>255)
<             fprintf(fdefine, "/* token %s is %d */0,
<                 tokset[i].name,
<                 tokset[i].value);
<     } else {
<         for( ; (c= *cp)!=' '; ++cp ) {
<             if(islower(c) || isupper(c) || isdigit(c) || c=='_')
<                 ;
<             else break;
---
>     if( *cp == ' ' ) ++cp; /* literals */
>
>     for( ; (c= *cp)!=' '; ++cp ){
>
>         if( islower(c) || isupper(c) || isdigit(c) || c=='_'); /* VOID */
>         else goto nodef;
660,663c471,475
<     if( c==' ' )
<         fprintf( fdefine, "# define %s %d0,
<                 tokset[i].name, tokset[i].value );
<     else error("Invalid token name");
---
>
>     fprintf( ftable, "# define %s %d0, tokset[i].name, tokset[i].value );
>     if( fdefine != NULL ) fprintf( fdefine, "# define %s %d0, tokset[i].name, tokset[i].value );
>
>     nodef:     ;
665c477

```

```

<  }
---
>
666a479
>
707,709c520
<  /* need to return '<' as such due to chain rules
<  case '<':
<      error("Type specification no longer needed");
---
>  case '<': /* get, and look up, a type name (union member name) */
725d535
<  */
730c540
<      tokname[0] = (c=="")?' ':'';
---
>      tokname[0] = ' ';
732c542
<      for(;;) {
---
>      for(;;){
739,740c549
<          if( ++i >= NAMESIZE )
<              error("Token too big");
---
>          if( ++i >= NAMESIZE ) --i;
744,745c553
<          if( ++i >= NAMESIZE )
<              error("Token too big");
---
>          if( ++i >= NAMESIZE ) --i;
747,773d554
<          if ((c = getc(finput))== '<') { /* specify lenv */
<              int j, bpoint;
<              tokname[i]=': ';
<              if( ++i >= NAMESIZE )
<                  error("Token too big");
<              bpoint=i;
<              while ((c=getc(finput))!='0' && c!='' &&
<                  c!=' ' && c!=EOF) {
<                  tokname[i]=c;
<                  if( ++i >= NAMESIZE )
<                      error("Token too big");
<              }
<              if (tokname[i-1]!='>')
<                  error("Unterminated whitespace environment tag");
<              tokname[i-1]=' ';
<              ungetc(c, finput);
<              for (j=0; j<nwhites; j++)
<                  if (!strcmp(tokname+bpoint,whiteset[j]))
<                      break;

```

```

<         if (j>=nwhites)
<             error("mis-specified whitespace env");
<     } else {
<         tokname[i]=':.';
<         if( ++i >= NAMESIZE )
<             error("Token too big");
<     }
<     ungetc(c, finput);
803,804c584
<     /* no longer possible to have '.' in id */
<     else if( islower(c) || isupper(c) || c=='_' || c=='$' ){
---
>     else if( islower(c) || isupper(c) || c=='_' || c=='.' || c=='$' ){
806c586
<         while( islower(c) || isupper(c) || isdigit(c) || c=='_' || c=='$' ){
---
>         while( islower(c) || isupper(c) || isdigit(c) || c=='_' || c=='.' || c=='$' ){
825,826d604
<         if( !strcmp(tokname,"whitespace")) return( WHITESPACE );
<         if( !strcmp(tokname,"tokprec")) return( TOKPREC );
832c610
<         /* if( !strcmp(tokname,"union")) return( UNION ); */
---
>         if( !strcmp(tokname,"union")) return( UNION );
837,841d614
<     c = getc(finput);
<     strcpy(swtag, wtag);
<     wtag[0]=' ';
<     if (c=='<') {
<         int i;
843,848c616
<         for (i=0; (c=getc(finput))!='>' && c!=' ' && c!='0' && c!=EOF; i++)
<             wtag[i]=c;
<         wtag[i]=' ';
<         if (c=='>') c=getc(finput);
<         else error("Unterminated whitespace tag");
<     }
---
>     c = getc(finput);
851c619
<     else if( c == '/' ) { /* look for comments */
---
>     else if( c == '/' ){ /* look for comments */
853c621
<     }
---
>     }
855,860c623,624
<     }
<     if( c == ':' ) return ( C_IDENTIFIER );
<     if( c == '"' ) {

```



```

<         cpyact(0, 2, 0);
<         return( C_IDENTIFIER );
<     }
---
>     }
>     if( c == ':' ) return( C_IDENTIFIER );
862d625
<     strcpy(wtag, swtag);
867,868d629
<     return t>=NTBASE?2:1;
< /*
875d635
< */
881c641
<     if (s[0]==' ' || s[0]=='')t=0;
---
>     if (s[0]==' ')t=0;
883c643
<         if(!strcmp(s,tokset[i].name)) {
---
>         if(!strcmp(s,tokset[i].name)){
898,902d657
< strcmp(a, b)
< {
<     return strcmp(a,b);
< }
<
907c662
<     /* fprintf( ftable, "0 line %d
---
>     fprintf( ftable, "0 line %d
946c701
<     /* fprintf( ftable, "0 line %d
---
>     fprintf( ftable, "0 line %d
969,971c724,726
<         while( c == '*' )
<             if( (c=getc(finput)) == '/' )
<                 return( i );
---
>         while( c == '*' ){
>             if( (c=getc(finput)) == '/' ) return( i );
>         }
974c729
<     }
---
>     }
976c731,732
< }
---
>     /* NOTREACHED */

```

```

>     }
978,980c734
< cpyact(offset, eval, rewind)
< long     rewind;
< { /* copy C action to the next ; or closing */
---
> cpyact(offset){ /* copy C action to the next ; or closing */
982,984d735
<     int  begst, brac2, incloop, outer=1;
<     FILE *outf;
<     long thisplace;
986,1000c737
<     if (rewind) {
<         thisplace=ftell(finput);
<         fseek(finput, rewind, 0);
<     }
<     if (eval==1) {
<         outf=fclasses;
<         fprintf(outf, "starteval0);
<     } else if (eval==2) {
<         outf=fclasses;
<         fprintf(outf, "startdecl0);
<     } else {
<         outf=faction;
<     }
<     /* if (!eval) fprintf( outf, "0 line %d
<     */
---
>     fprintf( faction, "0 line %d
1003,1005d739
<     brac2=0;
<     begst=1;
<     incloop=0;
1012,1033d745
< case '[':
<     if (!incloop && begst && eval) {
<         incloop=1;
<         fprintf(outf, "0* loop over children */0);
< fprintf(outf, "for (int yyloop=0; yyloop<numchildren(); yyloop+=cloop) {0);
<     } else {
<         ++brac2;
<         goto lcopy;
<     }
<     c = getc(finput);
<     goto swt;
< case ']':
<     if (incloop && !brac2 && eval) {
<         incloop=0;
<         begst=1;
<         fprintf(outf, "00* endcloop*/0);
<     } else {

```

```

<         --brac2;
<         goto lcopy;
<     }
<     c = getc(finput);
<     goto swt;
1035d746
<     begst=1;
1037,1041c748
<         putc( c , outf );
<         if (eval==1) fprintf(outf, "Ondeval0);
<         else if (eval==2) fprintf(outf, "Onddecl0);
<         if (rewind)
<             fseek(finput, thisplace, 0);
---
>         putc( c , faction );
1047d753
<     begst=1;
1049,1052d754
<     if (outer && eval) {
<         outer=0;
<         goto loop;
<     }
1060d761
<     error("$<ident> clauses no longer in use");
1067c768
<     fprintf( outf, "yyval");
---
>     fprintf( faction, "yyval");
1070c771
<         fprintf( outf, ".%s", typeset[tok] );
---
>         fprintf( faction, ".%s", typeset[tok] );
1074,1111d774
<     if ( isalpha(c) || c=='"' || c == "'") {
<         int  t, x, k, m, savek;
<         char  errmsg[100];
<
<         k=savek=1;
<         ungetc(c, finput);
<         t = gettok();
<         x = chfind(2, tokname);
<         if (absym(x))
<             error("$ reference to constant or action");
<         c = getc(finput);
<         if (c=='#') { /* more than one of it in prod */
<             k=0;
<             c=getc(finput);
<             while( isdigit(c) ) {
<                 k= k*10+c-'0';
<                 c = getc(finput);
<             }

```



```

>         if( j+offset <= 0 && tok < 0 ) error( "must specify type of %d", j+offset );
>         if( tok < 0 ) tok = fdtype( prdptr[nprod][j+offset] );
>         fprintf( faction, "%.s", typeset[tok] );
>     }
1152,1153c799,800
<         putc( '$' , outf );
<         if( s<0 ) putc( '-', outf );
---
>         putc( '$' , faction );
>         if( s<0 ) putc( '-', faction );
1157d803
<         begst=1;
1159,1163c805
<         if (!eval) putc( c, outf );
<         if (eval==1) fprintf(outf, "Ondeval0);
<         else if (eval==2) fprintf(outf, "Onddecl0);
<         if (rewind)
<             fseek(finput, thisplace, 0);
---
>         putc( c, faction );
1168c810
<         putc( c , outf );
---
>         putc( c , faction );
1170,1173c812
<         if( c != '*' ) {
<             begst=0;
<             goto swt;
<         }
---
>         if( c != '*' ) goto swt;
1177c816
<         putc( c , outf );
---
>         putc( c , faction );
1181c820
<             putc( c , outf );
---
>             putc( c , faction );
1184c823
<         putc( c , outf );
---
>         putc( c , faction );
1198d836
<         begst=0;
1200c838
<         putc( c , outf );
---
>         putc( c , faction );
1204c842
<             putc( c , outf );

```

```

---
>          putc( c , faction );
1210c848
<          putc( c , outf );
---
>          putc( c , faction );
1217,1219d854
< case ' ':
< case "":
<     goto lcopy;
1222d856
< default:  begst=0;
1227c861
<     putc( c , outf );
---
>     putc( c , faction );
diff myacc/y3.c oyacc/y3.c
10,11d9
< int g2debug = 1;
< int pkdebug = 0;
17,19d14
<     extern FILE*loutput;
<     extern char *whiteset[NTYPES], *whitespaces[NTYPES];
<     extern int  nwhites;
23,25d17
<     for (i=0; i<nwhites; i++)
<         fprintf(loutput, "%s%s0, whiteset[i], whitespaces[i]);
<     fprintf(loutput, "%%%0);
78c70
<     fclose(loutput);
---
>
80a73
> int pkdebug = 0;
182a176
> int g2debug = 0;
244,245c238
<         if( foutput != NULL )
<             fprintf( foutput, "74d: shift/reduce conflict (shift %d, red'n %d) on %s",
---
>         if( foutput != NULL ) fprintf( foutput, "74d: shift/reduce conflict (shift %d, red'n %d) on %s",
298,306c291
<     TLOOP(p) {
<         if( temp1[p]+lastred == 0 ) {
<             temp1[p]=0;
<             if (lastred &&
<                 strcmp(symnam(p),"Send") &&
<                 strcmp(symnam(p),"error"))
<                 makelexeme(p, i, lastred, 0);
<         }
<     }

```

```

---
> TLOOP(p) if( temp1[p]+lastred == 0 )temp1[p]=0;
366,368c351
<         else {
<             fprintf( foutput, "shift %d", j1 );
<             makelexeme(j0, i, j1, 1);
---
>         else fprintf( foutput, "shift %d", j1 );
370,372c353
<     } else {
<         fprintf( foutput, "reduce %d",-j1 );
<         makelexeme(j0, i, -j1, 0);
---
>     else fprintf( foutput, "reduce %d",-j1 );
374d354
<     }
378,380c358,359
<     if( lastred ) {
<         fprintf( foutput, "0. reduce %d0, lastred );
<     } else fprintf( foutput, "0. error0 );
---
>     if( lastred ) fprintf( foutput, "0. reduce %d0, lastred );
>     else fprintf( foutput, "0. error0 );
390,406d368
<
< makelexeme(i, s, whatto, toshift)
< {
<     int  dnum;
<     extern FILE *loutput;
<     extern char sympref();
<
<     if (tokset[i].value<256)
<         fprintf(loutput, "%d%s%d%s%d%c0,
<             s, (symnam(i)[0]=='?'?
<             tokset[i].value,
<             toshift?"shift":"reduce",
<             whatto,sympref(i));
<     else fprintf(loutput, "%d%s%d%s%d%c0,
<             s, symnam(i), tokset[i].value,
<             toshift?"shift":"reduce", whatto,sympref(i));
< }

```