Lily: A parser generator for LL(1) languages

Timothy Topper Taylor

LILY: A PARSER GENERATOR FOR LL(1) LANGUAGES

BY

TIMOTHY TOPPER TAYLOR, 1961-

A THESIS

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

1987

Approved by

Thomas J. Sager (Advisor) J. R. Metzner

[illegible]

ABSTRACT

This paper discusses the design and implementation of Lily, a language for generating LL(1) language parsers, originally designed by Dr. Thomas J. Sager of the University of Missouri-Rolla. A method for the automatic generation of parser tables is described which creates small, highly optimized tables, suitable for conversion to minimal perfect hash functions.

An implementation of Lily is discussed with attention to design goals, implementation of parser table generation, and table optimization techniques. Proposals are made detailing possibilities for further augmentation of the system. Examples of Lily programs are given as well as a manual for the system.

ACKNOWLEDGEMENT

The author wishes to thank his advisor, Dr. Thomas J. Sager, whose original design of the Lily source language figured heavily in the final design discussed here and whose encouragement was indispensible to the author's morale. Thanks are also due to committee members, Dr. John R. Metzner and Mr. Darrow F. Dawson for their advice, time and support. Finally, the author wishes to extend his gratitude to Ms. Theresa Fitzsimmons, Mr. Ryan Fitzsimmons, Ms. Trisha Fitzsimmons, Mr. YiLi Sung, Ms. Catherine Chan, and, the author's parents, Mr. Ronald G. Taylor and Mrs. Sharon L. McCaw.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

I. INTRODUCTION

Lily is a system which assists in the automatic generation of compiler front ends. It was originally designed by Dr. Thomas J. Sager [1] to provide both a test application for his mincycle algorithm as well as a simple, effective parser generator for use in his compiler classes. Although the author has made substantial modifications to the original design, the system, in its final implementation, retains a great deal of the flavor of Dr. Sager's initial concept. Appropriately then, the name Lily has been retained, after Dr. Sager's daughter.

Lily is a relatively simple, readable parser generation tool that operates well when used in conjunction with Borland International's Turbo Pascal environment. This is desirable since Turbo Pascal enjoys a wide base of popularity and is, for all its faults, a fast, inexpensive, and easily understood programming facility. These factors make it an excellent choice for university students. Lily, which writes parsing functions in Turbo Pascal can serve as a teaching aid in a compiler course.

Lily comprises a prototype for what is hoped will become a larger-scale compiler-compiler project. Such a project, if the author's experience with Lily is any yardstick, would certainly be worthwhile for the participants. Moreover, the process of writing augmentations to Lily can be facilitated with the aid of Lily itself. Thus, the system can furnish the first link in a series of self-improvements.

In this chapter some of the background behind the creation of the Lily parser generation language is considered. The first section examines other compiler generating systems which have been implemented and gives a brief history of such systems. This should help to form the basis for a subsequent section reviewing the objectives used in the design of Lily. The last section of the chapter is devoted to a discussion of those objectives which have been realized in the current implementation.

A. <u>REVIEW OF EXISTING COMPILER-COMPILERS</u>

A compiler-compiler, or metacompiler, is a programming tool for automatically constructing compilers. Ideally, it is a program that, given a description of a language and a target machine, produces a fully functioning compiler. While the full realization of this ideal has proven elusive, significant strides have been made since metacompilers first began to emerge in the sixties.

Early work in the field, enduringly popular programs such as YACC and LEX, concentrated only on automating the development of the compiler's front end, the lexical and syntactic analysis phases. Later, more ambitious compiler-compilers began to appear to tackle the problems posed by context-sensitive semantic analysis and code generation, the compiler's back end. As of this writing, programs like

LINGUIST-86 and MUG2 have succeeded in automating the majority of the compiler writing process.

The first compiler generation programs focused on automating the development of compiler front ends. Most of these precursory systems, many still in use, accepted input in the form of a metalanguage such as the Backus-Naur Form (BNF) and produced tables for driving a standard parsing algorithm. Those parts of a compiler not easily amenable to formal specification, e.g. semantic analysis or code generation, were left to be programmed by the compiler implementor.

More modern systems use extended formal language descriptions such as attribute grammars to allow the automatic implementation of compiler back ends. These systems attempt to automate the creation of the more complicated aspects of a compiler such as semantic analysis, code generation, and code optimization.

1. Scanner and Parser Generation in Lex and YACC: LEX [2] is a generator for lexical analyzers (scanners). A LEX scanner is defined using a variation on regular expression notation to describe the token structure or low-level syntax of a language. Given a file of appropriate source code, Lex generates C language functions composed of a table and a standard, table-driven scanning algorithm.

Facilities called "hooks" are provided to allow the user to define C language routines to be executed at strategic points in a scanner's operation. For instance, a user who wished to verify that a sequence of digits, recognized by a Lex-fashioned scanner, represent a value within a given range could use appropriate C language routines to accomplish this.

The concept of hooks is shared by YACC [3], usually considered a companion program to Lex. While Lex is used for writing lexical analyzers, YACC (Yet Another Compiler-Compiler) is used for writing parsers. Like Lex, YACC produces C subprograms based on a combination of tables and a standard, table-driven algorithm.

YACC, an implementation of LALR(1) parser theory, takes input in a form similar to BNF. In another resemblance to Lex, YACC provides hooks, interfacing the operation of the generated parser to C language routines which the user may wish to employ. These routines may effect the operation of the parser, for example, to resolve non-LALR(1) ambiguities. Another important use of hooks in YACC is the creation of an intermediate language representation of the source code supplied to the generated parser. Thus, a YACC-generated function might check for correct context-free syntax and then, using hooks to hand-coded actions, build and consult a symbol table or assemble an abstract syntax tree.

2. Attribute Grammars: A perusal of the literature on newer compiler generation systems quickly underscores the importance of attribute grammars (AG's) in the modern

metacompiler. Some of the more recent systems include: LILA [4] (Language Implementation Laboratory), PQCC (Production-Quality Compiler-Compiler), Linguist-86, GAG [5] (Generator based on Attribute Grammars), and MUG2. All of these systems use attribute grammars, in one form or another, to specify compiler characteristics. Attribute grammars, like BNF, constitute a formalism which can effectively form the basis of a metacompiler.

An attribute grammar, described by Knuth [6], consists of three parts: a context-free grammar (CFG), a set of attributes, and a set of functions over the attributes. Productions are written in standard BNF or some suitable extension. Then, with each symbol of the grammar, attributes, constituting semantic information, are associated. Each attribute may take on a possibly infinite set of values comprising a range of <u>attribute instances</u>. For example a CFG symbol for a variable might include attributes for a type, a scope, and an address descriptor.

The set of attributes in an attribute grammar is partitioned into two subsets: the <u>synthesized</u> attributes and the <u>inherited</u> attributes. Synthesized attributes take on values which are to be passed up the parse tree toward the root. Inherited attributes, on the other hand, take on values passed down the parse tree or between siblings.

The functions of an attribute grammar serve to define the synthesized attributes of language symbols in terms of other attributes. One problem that can occur in the functional definition of a synthesized attribute is the introduction of oriented cycles; i.e. an attribute is defined in terms of itself through some series of functional dependencies. An attribute grammar is termed well-ordered if no such cyclic dependencies exist.

3. Semantic Analysis in Linguist-86: Linguist-86 [7], mentioned above, is a good example of a metacompiler system which employs attribute grammars. This translator writing system has been used to develop a production compiler for Intel's Pascal-86. Figure 1 shows an example of the coding of an attribute grammar in the Linguist-86 system. Note in the example, a representation of Ada based numbers, the juxtaposition of attributes (the identifiers following the periods) and attribute-evaluating assignment functions on a skeletal context-free grammar.

A program called the Semanticist uses this attribute grammar description to construct an alternating-pass attribute evaluator for the compiler being implemented. At compile time this evaluator constructs an intermediate-level form of the source language called an attributed parse tree (APT), essentially a parse tree in which the symbol nodes contain fields for attribute instances.

The actual computation of attribute instances is accomplished by making alternating, left and right, depth-first traversals of the APT. As each instance is evaluated it becomes available as a parameter to functions evaluating

```
number ::= digits1 '#' digits2
  number.VAL          = digits2.VAL
  digits2.RADIX       = digits1.VAL
  digits1.RADIX       = 10
  digits1.POWER       = 1
  digits2.POWER       = 1

digits ::= digit
  digits.VAL  = digit.VAL
  digit.POWER = digits.POWER

digits0 ::= digits1 digit
  digits0.VAL      = digits1.VAL + digit.VAL
  digits1.RADIX    = digits0.RADIX
  digits1.POWER    = digits0.POWER * digits0.RADIX
  digit.POWER      = digits0.POWER

digit ::= '0'
  digit.VAL = 0

digit ::= '1'
  digit.VAL = 1 * digit.POWER

digit ::= '2'
  digit.VAL = 2 * digit.POWER

          .
          .
          .

digit ::= 'F' | 'f'
  digit.VAL = 15 * digit.POWER
```

Figure 1. Example of an attribute grammar in Linguist-86 [7]*

* Typographical errors corrected by the author

other instances in the tree. The number cf passes necessary to evaluate all attribute instances in the APT depends on the functional dependencies of the attributes in the AG.

4. Code Generator Generation: A number of different approaches have been made to the problem of automatic code generator generation. Ganapathi et al. [8] divide these methods into three categories: interpretative, pattern-matched, and table-driven. One of the most successful methods has been the table-driven code generation system introduced by Graham and Glanville.

The Graham-Glanville system is designed to generate code generators capable of producing efficient code on a wide variety of machines and architectures. This system focuses on the compiler back end and leaves the front end to the responsibility of the compiler implementor. Before attempting to generate a code generator, such compiler issues as syntax, semantic analysis, and variable binding must be resolved.

The front end of the compiler creates a prefix instruction, intermediate representation (IR) which is passed to the automatically generated code generator. The code generator has been created by associating target machine language sequences with IR structures. This association is made using context-free language definitions specifying an IR structure suffixed by the target machine code equivalent. The code generator attempts to parse strings of IR emitting appropriate code in the process. Whenever a string of IR matches more then one production structure, heuristic methods based on, for example, code efficiency and size, resolve the ambiguity.

A variation on the Graham-Glanville approach has been applied in the Production-Quality Compiler-compiler System (PQCC) [9]. PQCC uses a different form of IR, a tree-based structure called TCOL. Instead of matching prefix IR instructions by using context-free language tools, PQCC associates patterns of tree templates (sub-trees) with target machine code sequences. This method attempts to generate code by recursively matching templates to all of the sub-trees of the TCOL program representation. Such a "tiled" template fully represents the code to be generated.

5. Code Optimizer Generation in MUG2: The translator writing system MUG2, described by Ganzinger et al. [10], uses a language called OPTRAN to define code optimizer modules. An OPTRAN program consists of a series of AT-rules (attribute translation rules) to specify mappings from less efficient structures to more efficient structures. While OPTRAN makes it difficult to specify, for instance, optimizations based on register allocation or preferred machine code instructions, it does support structural optimizations such as constant folding and loop optimization.

An AT-rule consists of four parts: an input template, a predicate over the attributes of the intermediate language representation of a program, a set of attribute functions, and an output template. If the predicate is true then the generated optimizer replaces the intermediate structure matched by the input template to the intermediate structure matched by the output template. The attribute functions map attributes from the old to the new structure.

In MUG2, the intermediate language is a tree-like structure similar to the TCOL of PQCC and the attributed parse tree of Linguist-86. It is composed of operator nodes, operand leaves, and attribute fields. The input and output templates of an AT-rule are language representations of sub-trees in the intermediate language. Thus, an AT-rule for a constant-folding optimization has an input template consisting of an operator node and two constant leaves while the output template consists merely of a single leaf. This leaf will carry a single attribute which is the result of applying the operator to the attributes of the input template leaves. Figure 2, adapted from Ganzinger et al., shows a diagram of the AT-rule for a constant-folding optimization.

B. OBJECTIVES FOR LILY

Lily, in its current stage of development, has objectives similar to those of YACC and LEX. The implementation that is the subject of this paper has been provided with a variety of features some of which were foreseen at the outset while others came to light only in the course of

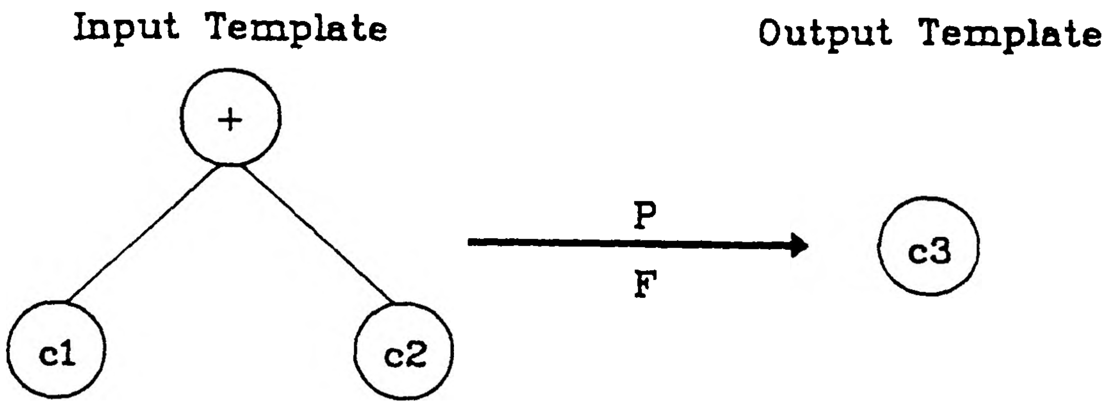

Figure 2. Diagram of constant-folding AT-rule*

* The input template shows a sum of two constants. The output template shows a single constant which can be replaced for the sum. The predicate P, in this case would be equivalent to T, i.e. always true. Thus, two constants may always be folded when added as shown. The function F would serve to add the value attributes of the two constants, c1 and c2 to arrive at the value attribute of constant c3.

development. Quite a number of these features were due to Dr. Sager's original conception of the system. A list of the design objectives for Lily, both original and acquired, includes the following:

1. Simplicity-- Compiler specifications written in Lily should be easy to write, comprehend, and maintain.

2. Expressive Power-- The Lily program should be able to generate parsers for languages with grammars of reasonable size. In the event that a grammar is ambiguous, methods for disambiguation, possibly interactive, should be provided.

3. Space and time efficiency-- Lily should operate quickly to allow for quick reprocessing of input once error corrections have been made. The parser functions generated by Lily should be fast and compact.

4. Generality-- Like YACC and LEX, Lily should generate subprograms applicable to tasks other than compiler generation.

5. Convenience-- The user should need to know as little as possible about the internal functioning of Lily and its generated subprograms. The language itself should provide for terse definitions while maintaining clarity. Standard features should be included where they are of sufficient usefulness.

6. Formality of language definitions-- Lily should provide attribute translation as a method of specifying context-sensitive syntactic structure (program semantics) as well as less formal methods such as the use of action symbols.

7. Correctness of operation-- Most important was that the Lily program should operate correctly under every conceivable condition as should the subprograms generated.

These and other less significant goals such as readable, informative screen displays, and convenient run-time interaction with the user were at the foundation of the project.

C. REALIZATION OF LILY OBJECTIVES

Most of the design objectives in the previous section were achieved to a reasonable extent. Invariably, some of the goals for the system were at odds either with one another or with the finite nature of the proposed time-frame for implementation. The largest sacrifice was the abandonment of support for formal attribute evaluation. The magnitude of such an implementation was quickly recognized and the scope of the project was narrowed to a less ambitious dimension. This permitted the author to concentrate more fully on the other, more tractable portions of the project.

As a result, the semantic analysis and code generation phases of a Lily-generated compiler must be carried out, as in YACC, by hand-coded procedures (actions) provided by the user. Substantial opportunity for research and development in the area of attribute evaluation for Lily remains for any who might care to carry the project farther. The author of this paper will confine himself merely to suggestions on this score in the pages that follow.

Apart from the early exclusion of attributed nonterminals, the other goals were largely achieved, starting with simplicity. The Lily parser specification language is very compact containing only thirty-eight different tokens in its token set and fifteen key words. Each parser specification contains only a few distinct parts in a standard order. Several of these parts are optional and are included to give the user more expressive power.

With regard to expressive power, Lily may be used to generate LL(1) language parsers of a reasonable size, up to four hundred and ninety (490) distinct grammatic symbols. Languages which are LL(k), i.e. containing production alternatives with finite-length prefixes, may also be generated subject to the requirement that the user provide code for ambiguity resolution. This process is facilitated in Lily by allowing the user to request look-ahead buffering for any generated parser.

One of the most important factors in Lily, reasonable space-time efficiency in both the metacompiler and the generated parsers, has been substantially achieved in this implementation. While the metacompiler does employ space-wasting bit maps (Pascal sets) it also makes use of automatic overlaying to achieve efficiency in the code segment. Further, the metacompiler code assigns a high priority to execution speed. In most cases this will allow the user to re-metacompile quickly after correcting errors.

The subprograms generated by Lily, while large, are designed to facilitate the use of overlays; all important variables are maintained at the global level. In addition, execution speed was enhanced by encoding tables directly as Turbo Pascal "typed constants." This eliminates the necessity of initializing tables by reading them from disk files. In-line machine code is used to facilitate the selection of actions. Used in place of the less efficient case structure, this code simulates a compiled version of the "computed goto" of BASIC.

Generality was achieved by not limiting the types of data which may be processed using a Lily-generated subprogram. The standard low-level parsers supplied with Lily read data from files on a byte-by-byte basis. This means that Lily could possibly be used to generate such programs as unassemblers and, by redirecting input, screen drivers. These applications are in addition to the standard use of Lily, generating compiler front ends. An unusual use of Lily to generate "graftal" tree [11] graphics has been implemented by the author; the code for this is shown in

Appendix B, figure 17. A more usual implementation is shown in Appendix B, figure 15, a Lily-coded parser for the language Nothing, a language created by Dr. Sager for a compiler structure course.

The source language of the Lily metacompiler is designed to allow multiple parsers to be defined while providing, in addition, a means for automatic interfacing between parsers. The structure of the language is given in the grammar of Appendix A. Appendix B consists of a manual describing the meaning and use of the language. It is recommended that the reader browse through this manual to gain a better understanding of the system.

Productions are encoded using a form of regular right part grammar (RRPG). This method allows for a more readable grammar specification since recursion need not be used to specify certain kinds of repetitive structures. At the same time, the elimination of unnecessary recursion from grammar definitions removes a run-time burden from the generated parser; stack operations are significantly curtailed. A final reason for using this method is that it permits a somewhat uniform way of representing both parsers and their lexical analyzers. In Lily, a lexical analyzer is just another parser with multiple goal symbols for tokens. To quote LaLonde [12] on the advantages of RRPG's:

> "...when restricted to [conventional] CF grammars, we are forced to use a recursive definition [for structures not intuitively recursive]. Anyone learning the language via a CF grammar must therefore be able to distinguish between recursion which is an inherent property of the language and

recursion which is introduced as a consequence of an inadequate descriptive mechanism.
It is...clear that regular languages can be described more easily with RRPGs than with CFGs. RRPG descriptions, however, (unlike that of CFGs), can be used directly for constructing scanners. Thus scanners and parsers can be described and constructed in a uniform way; i.e. scanners can be thought of as restricted types of parsers."

Representing a lexical analyzer in the same way as a parser is, however, not without a drawback in Lily. Tokens of a Lily lexical analyzer must have unique prefixes in order to be LL(1). This means that, for instance, the token structures representing a colon and an assignment operator in Pascal (":" and ":=") are ambiguous and require ambiguity-resolving actions. A method for automatically resolving finite-length prefix ambiguities of this type is feasible but does not form a part of the current implementation. This method is discussed in the last chapter of this thesis where conclusions are drawn and further research possibilities are suggested.

Extensive validation of the metacompiler and the two hash function generation programs was done in order to help assure correct operation. Front ends for three complete test languages were written and debugged using the system. In addition, numerous error conditions were checked by supplying the programs with various erroneous data. This method uncovered many bugs which were systematically removed. Nevertheless, programs the size of those currently making up the Lily system are likely to have unexpected bugs. It is hoped that information provided by the programs themselves will be of assistance in diagnosing and tracking down errors.

II. METHODOLOGY

This chapter discusses the methodology used in implementing the author's parser generator. The two other programs which complete the system are due to Dr. Sager and were modified by the author to conform to the standards of the parser generator. These two programs are used to convert tables generated by the parser generator to minimal perfect hash functions. For an explanation of the theory behind minimal perfect hash functions the reader is referred to Dr. Sager's paper on the subject [13].

The Lily parser generator was constructed in three overlapping phases. First, a recursive descent parser was written. This was followed by code for computing relations over parser grammar symbols. Finally, procedures were added to generate the required tables and Pascal functions.

The design of the parser generator language began with an uncertain modification of the original Lily design by Dr. Sager. Most of the desired features were known ahead of time but others only became apparent in the course of the implementation. For instance, the idea of a language allowing multiple parser definitions was clear from the start. A convenient method of interfacing the defined parsers with one another was, however, an unsettled matter. At the outset of the project, not even the form of the output was fully known.

Gradually, as the project progressed, necessity forced the develepment of a concise program structure. Earlier ideas (a now-extinct wild-card character for instance) were seen to be either unnecessary or intractable given the scope of the project. At the same time, a clearer view of the problem gave way to solutions for old difficulties and helped to suggest unanticipated improvements. In this way the parser generator program grew by increments from a trepid thousand-line Turbo Pascal program to a much more capable program of some forty-seven hundred lines.

This chapter explains some of the methodology used in writing the parser generator of Lily. While it does not purport to be a comprehensive log of the project, it does provide an overall examination of the techniques that were applied and which now operate in the finished program.

A. STRUCTURE OF THE PARSER GENERATOR

1. Parser: The parser of the Lily parser generator is a conventional recursive descent parser of the type recommended by Wirth [14]. Source program semantics are checked by ad hoc routines embedded in the parser code using a syntax-directed approach. Warning-level and fatal error messages are written both to the screen and to a disk file. Tables XII and XIII in appendix B list the possible messages and their respective meanings.

In addition to detecting errors, the parser is responsible for constructing a symbol table to represent the parser specifications in the source code. This table contains information relevant to the table-construction phase of the parser generator. The grammar of each parser is represented by a list of trees, each tree embodying a production. Table I shows the contents of parser specification records in the symbol table.

2. <u>Table and Function Constructors</u>: The process of table and function construction is accomplished working strictly with the symbol table generated by the parser; Lily creates this table in a single pass over the source. Code for the table construction phase overlays the code for the parser. Table and function processing, then, proceeds one parser specification at a time.

The first step in constructing a parsing function table for a parser is to enumerate the symbols in the grammar structure. This permits the manipulation of symbols using sets and relations instead of exclusively referencing the list structure in the symbol table. Enumerating the symbols in this way yields a grammar representation suited to the application of well-known, set-based algorithms for computing relations over the grammatic symbols.

The next step involves the isolation of nullable nonterminals, those nonterminals of the grammar capable of generating the empty string. A relatively simple recursive function isolates the nullable nonterminals by repeatedly checking the grammar for null right sides. Parenthetic enclosures are treated, as will be explained, like nonterminals, with the zero-or-more-repetitions enclosure being

TABLE I

INFORMATION IN A SYMBOL TABLE ENTRY

Field	Meaning
name	name of parser
local_decl	user local definitions
fore	user fore definitions
aft	user aft definitions
receives_name	name of parser received by current parser
reattribs	attributes of received parser
regoals	goals of received parser
rekeywords	key words of received parser
renum_tokens	number of tokens of received parser
look	amount of look-ahead in current parser
goals	goals of current parser
keywords	key words of current parser
key_type	type of key words of current parser
sets	list of sets in current parser
sets_begin	ordinal assigned to first set
grammar	list of trees containing grammar of current parser
actions	list of actions of current parser
disambig	list of disambiguations of current parser
null_nonterms	set of nullable nonterminals
num_tokens	number of tokens output by the current parser
next	pointer to next parser record

inherently nullable. Action symbols and the empty string symbol are considered syntactically equivalent. In order to determine the nullable nonterminals a function traverses the grammar tree searching all paths to determine which ones generate the null string.

After finding the nullable nonterminals, well-known methods are used to determine director symbols, those terminal symbols which may occur first in deriving a given nonterminal. Knowlege of the director symbols for all nonterminals makes it possible to check a grammar for correctness as well as construct the needed parser. Extensions to the common methods were required because Lily's grammar notation is not linear.

Armed with the director symbols, the process of table construction concludes by writing a Pascal function and parsing function table for the current parser specification. Each Pascal function generated consists of a single basic structure with tables, named entities and certain code fragments varying slightly from parser to parser.

Various forms of error checking take place during function and table generation: the grammar is checked for the existence of left recursion; nonterminal symbols are checked for usefulness; the size of the grammar is checked; and, finally, nondeterminism is detected whenever language constructs are not LL(1). If the process concludes without error messages or warnings, then a correct parser table and driving function should result.

B. REGULAR RIGHT PART GRAMMARS

Lily productions are defined using a form of regular right part grammar (RRPG) similar to common so-called extensions of BNF. RRPG's are desirable because they permit a more compact, readable production definition than conventional linear grammars. Grammar constructs which are repetitive but not inherently recursive may be described succinctly and conveniently using RRPG's whereas linear grammars often require the use of more cumbersome recursive expressions. Consider, for example, the productions below.

```
1.  ConstPart::= const ConstDecl {ConstDecl}
    ConstDecl::= ident = number ;

2.  [const_part]: const, (ident, equal, number,
       semicolon)+;

3.  ConstPart::= const ConstDeclList
    ConstDeclList::= ConstDecl ConstDeclList
    ConstDeclList::= empty
    ConstDecl::= identifier = number ;
```

All three sets of productions define the same structure, a Pascal-like constant declaration part. (Assume the terminal symbols in all three are provided by some lexical analyzer.) The first set is expressed in a common RRPG extension of BNF allowing braces to specify zero or more repetitions of an enclosed structure. This familiar extension is used (somewhat carelessly) to describe the syntax of Turbo Pascal in the compiler manual.

The second set of productions conveys the same constant part construct using Lily RRPG notation. Lily notation provides enclosures signifying: zero or more repetitions ("(" and ")*"), one or more repetitions ("(" and ")+"), and exactly one repetition ("(" and ")"). Notice that the use of the enclosures "(" and ")+", indicating one or more repetitions, reduces the number of symbols needed for the production set in the example.

In the third set of productions the original form of BNF is used. It can be seen that this representation is the least adequate of the three; the number of productions is larger, the number of symbols is larger, and, confusingly to the eye, the use of recursion is necessary.

Lily productions contain two operators, the comma (","), and the sheffer stroke ("|"). The comma symbolizes the operation of concatenation while the sheffer stroke represents alternation. In other extensions of BNF, concatenation is assumed when two symbols are separated by a space. However, because Lily productions may contain a large and varied character set, concatenation has been made explicit for the sake of clarity.

In the remainder of this thesis, most productions are defined using the Lily RRPG style. This seems far less clumsy than attempting to provide extended BNF descriptions side-by-side with Lily equivalents. Most of the productions used are self-explanatory. The reader experiencing difficulty is referred to Appendix B for a complete description of Lily production notation.

C. CONVERTING LILY GRAMMARS TO PARSING FUNCTIONS

1. Determining Director Symbols: In order to determine various facts about a grammar, Lily employs relations based on Knuth [15] and Tremblay-Sorenson [16]. With these relations in hand it is possible to derive sets useful for determining whether or not a grammar is LL(1). Instances of nondeterminism, as well as left-recursion can be isolated with moderate computational effort.

In this discussion a CFG is represented by a quadruple (V, T, S, P) where: V is a set of variables (nonterminals); T is a set of terminal symbols; S, in V, is the start symbol; and P is a set of productions.

FIRST and FOLLOW sets are of importance in developing an LL(1) parsing table for a CFG. Each symbol X in (V + T) is associated with two sets, FIRST(X) and FOLLOW(X). FIRST(X) contains all terminal symbols which may be encountered first in a derivation of X while FOLLOW(X) contains all terminal symbols which may properly follow a derivation of X. These two sets comprise what Griffiths [17] refers to as director symbols. They are applied in Lily-generated parsers to decide when nonterminal transitions are made.

The computation of the FIRST sets is effected using an intermediate relation F. This relation can be thought of as an "immediate" first relation. Consider the CFG production where N is a nonterminal and X_i is any terminal or nonterminal grammar symbol: $N ::= X_1\ X_2 \ldots X_n$.

Then, $N\ F\ X_1$ and, if X_1 is a nullable symbol then

$N\ F\ X_2$. If X_2 is also nullable then $N\ F\ X_3$ and so on. Trivially, for each terminal symbol X_i, including action symbols and the empty string, $X_i\ F\ X_i$, for all i from 1 to n.

Thus the F relation defines all first-derivation paths between nonterminal symbols in left sides and symbols in their corresponding right sides. Using Warshall's Algorithm to form F^+, the transitive closure of F, identifies all first-derivation paths between grammar symbols.

This transitive closure may be used to isolate left-recursion in the grammar. A grammar is left-recursive in a nonterminal N if and only if $N ::=^+ N\ X_1\ X_2\ \ldots\ X_n$; that left-recursion is implied whenever a nonterminal produces, in one or more steps, a string of which the nonterminal is a prefix. Thus if $N\ F^+\ N$ for any nonterminal N, then the grammar is left-recursive in N.

Lily uses this condition to determine whether or not a grammar is left-recursive. Both Griffiths and Knuth note that the LL(1) condition, that director symbols of alternative derivations be disjoint, encompasses the prohibition against left-recursion. Since Lily checks for the LL(1) condition, the use of F^+ to detect left-recursion is somewhat redundant. However, as Griffiths points out, the use of the transitive closure does allow for more meaningful diagnostic messages.

The FIRST sets for the grammar symbols are taken directly from the F^+ relation. Lily implements a relation as an array of sets indexed by the ordinals corresponding to grammar symbols. By removing nonterminal elements from each set in the F^+ array, an array of FIRST sets is formed in place, constituting a FIRST relation. This method saves on space since it is unnecessary to maintain the FIRST and FOLLOW sets as distinct entities from the relations used to compute them.

The FOLLOW sets are calculated using the FIRST relation in addition to two other intermediate relations, L and B. For a production $N ::= X_1 X_2 \ldots X_n$, as above, let X_n L Further, if X_n is nullable, then let X_{n-1} L N, and if X_{n-1} is nullable then let X_{n-2} L N and so on. The L relation may be considered a "last of" relation. L^* is computed using Warshall's algorithm to form L^+ and then effectively setting all bits along the main diagonal.

The B relation, mnemonically the "before" relation, can be computed by letting X_i B X_{i+1} for all i from 1 to n-1. Further, if X_{i+1} is nullable, then let X_i L X_{i+2}. If X_{i+2} is nullable let X_i L X_{i+3}, and so on.

FOLLOW sets are then calculated by forming the composite relation $(L^*)(B)$(FIRST). This relation is composed of an array of FOLLOW sets. This method differs slightly from Tremblay and Sorenson who form the composite $(L^*)(B)(F^*)$. However, since nonterminal references are removed from this composite, the effect is identical.

It should be noted that the RRPG notation of Lily and the addition of action symbols and sets requires the extension of the method given above. Such an extension is straightforward since all Lily grammars are implicitly rewritten in a form corresponding to the form used above.

Action symbols are defined to be syntactically equivalent to the null string. That is, an action symbol in a production alternative has the same effect on the parser as the null string. The difference lies in the fact that an action symbol causes the parser to perform some specified user action after the parser has executed the corresponding transition.

An alternative list inside the metabrackets "(" and ")" causes lily to insert an additional symbol treated as a nonterminal. This symbol takes a definition which can be written in standard form. Formally,

> (a_1 | a_2 | ... | a_n); is equivalent to [A], where
> [A] is defined by the production
>
> [A]: a_1 | a_2 | ... | a_n;

Here, as below, a_i stands for the ith alternative in a list of alternative phrases.

An alternative list inside the metabrackets "(" and ")+" is similarly treated by introducing a new symbol which is defined right recursively in standard form. Formally,

> (a_1 | a_2 | ... | a_n)+; is equivalent to [A], where
> [A] is defined by the productions:
>
> [A]: a_1, [A'] | a_2, [A'] | ... | a_n, [A'];
>
> [A']: [A] | #;

Note that this requires that no alternative, a_i, be nullable since this would introduce left-recursion. This is intuitively consistent with the meaning of these metabrackets; in a repeated nullable nonterminal, the repetitive recognition of the empty string results in an endless loop.

In addition, it should be noted that <u>two</u> symbols have been added instead of one. Lily maintains the first new symbol (in this case A) and discards the second. The second symbol is not necessary and its removal constitutes an optimization which is discussed later in this paper.

The last pair of metabrackets, "(" and ")*", are defined in much the same way as the preceding pair. Formally,

> (a_1 | a_2 | ... | a_n)*; is equivalent to [A], where
> [A] is defined by the production
>   [A]: a_1, [A] | a_2, [A] | ... | a_n, [A] | #;

Once again, no alternative, a_i, may be nullable as this would imply left-recursion.

Finally, set symbols, which represent groups of tokens, may be expanded to equivalent alternative lists. Formally,

```
[A]: a_set; can be rewritten as:
[A]: (t1 | t2 | ... | tn);
```

Here, t1, t2, ..., tn constitute all of the token elements of the set.

2. <u>Transition Networks</u>: The description of parsers generated by Lily can be facilitated by depicting the Lily-generated push-down automaton as its equivalent recursive transition network (see Woods [18]). RTN's have often been applied in parsing natural languages. They are useful because they are equivalent to context-free grammars in power while maintaining the convenience and perspicuity of finite automata graphs. An RTN is an ordered digraph having

arcs labeled with grammar symbols and nodes corresponding to parser states.

Consider figure 3 which shows a sample grammar and its representation as an RTN. The action of the parser consists of shifts, goto's, and push-goto combinations. When the

(a)

```
[Expr]: [Term], ('+', [Term])*;

[Term]: [Fact], ('*', [Fact])*;

[Fact]: '(', [Expr], ')' | 'i';
```

(b)

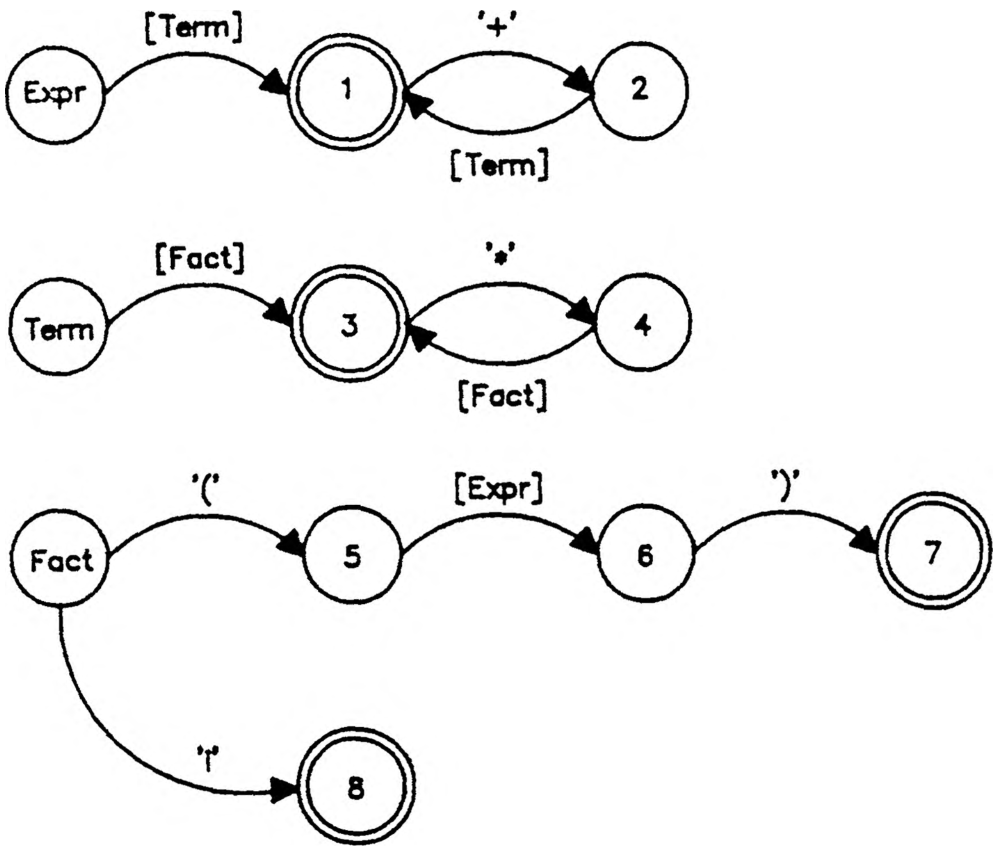

Figure 3. (a) A sample grammar in Lily notation
(b) A recursive transition network for sample grammar

parser is in a given state it attempts to change states by recognizing the language symbol on an outwardly directed arc. If the language symbol is a terminal symbol, then the parser changes states and shifts the input string. If the language symbol is a nonterminal symbol, then, on seeing a director symbol of the nonterminal, the parser pushes the state at the end of the arc and goes to the state named for the nonterminal. Finally, not shown in the figure, if the language symbol on an outgoing arc is an action symbol or null string symbol, the parser simply goes to the state at the end of the arc, executing any action symbol action prior to doing so.

The named states in the figure (Expr, Term, and Fact) are states assumed by the parser <u>before</u> seeing a string structure corresponding to the name. When the parser is in one of these states, it expects to see a string defined by a nonterminal. Under normal circumstances the parser will go to one of these states if a director symbol of the represented nonterminal is currently being scanned. At other times, default transitions may be made to these states when the parser is employing a Lily optimization. This optimization by defaults is discussed in chapter III.

Other states (those given numbers in the figure) are entered by the parser <u>after</u> it has recognized a structure. When the parser is in one of these states, the syntactic structure of the arc which lead to the state has already been processed. For this reason, care must be taken in the ordering of action symbols which require the use of the current token under scan. A good rule of thumb is: When an action requires the use of the current token, that action should be <u>prefixed</u> to the symbol for the token, as opposed to suffixed. This assures that the parser will execute the action prior to shifting past the needed token.

Doubly circled states (states 1, 3, 7, and 8 in the figure) are final states. If the parser is in a final state and no transition is possible, it pops the new state from the top of the stack. Whenever an empty stack is popped, the parser accepts the input string. On the other hand, if the parser is not in a final state and no transitions are possible, then the parser rejects the input string.

Woods notes that an RTN graph is a model of a push-down automaton which accepts on empty stack. The graph resembles the graphs depicting finite automata. Indeed, when productions of the corresponding grammar contain no nonterminal symbols, the RTN stack becomes unnecessary and the parser, as a result, becomes equivalent to a finite automaton.

3. <u>Parser Construction</u>: Lily constructs parsers according to the method above. Once the director symbols of grammar nonterminals have been isolated they may be used to make transition decisions between states. These transition decisions will comprise RTN arcs. If, from any state, all outward arcs are labeled with distinct director symbols, then the RTN is deterministic and the grammar is consistent

with the LL(1) condition; i.e. the director symbol sets for grammar alternatives are disjoint. (see Griffiths [17]).

Lily parsers are driven by two distinct tables: a table of transitions based on state-token pairs; and a table of default transitions based only on the parser state. The first table, called the _main parsing table_, encompasses transitions which are made only when the parser is in a given state, scanning a given token. The tokens which should be seen from a state consist of the director symbols of alternatives (outward arcs) of that state. The second table, called the _default table_, corresponds to transitions which are made if the current state-token pair fails to match an entry of the main parsing table.

The default table contains an entry for every state in the RTN. Final states have an entry which signals the parser to execute a pop. Other states have entries which either cause the parser to signal an error or make a transition on the empty string.

Representing the RTN with two tables instead of one is necessary only because small table size is of key importance to the effective generation of minimal perfect hash functions. These functions constitute a very efficient method of table look-up in generated parsers. The main parsing table is converted to such a function by the Lily system minimal perfect hash function generators. While the mincycle algorithm used in the generators is considered sound for creating tables of up to 512 entries, in Lily the method can become very time consuming for far fewer entries.

This is because the method relies on the relationship between keys as much as on the number of keys. Pseudo-randomness in the keys is necessary to minimize the amount of work done by the mincycle algorithm. Unfortunately, state-token pairs for a parser table are obviously not pseudo-random; a given state or token may occur many times in a parser table. Thus, a single, monolithic table can easily overtax the mincycle programs.

The two-table method resolves this difficulties as well as generating useful side effects. For instance, since the default table consists primarily of error entries, it is effectively quite sparse. However, it is possible to contravene these error entries, using Lily, to associate them with actions for error diagnostics or recovery. In addition, optimization techniques can be applied yielding two tables which, together, consume space comparable to that of a single monolithic table.

Construction of the tables is accomplished by referring to the grammar to determine the states and arcs of an RTN. Lily supports two types of grammars, single and multiple goal, which are intuitively different but effectively identical. The start state of an RTN for a single goal grammar corresponds directly to the goal symbol of that grammar. In slight contrast, the start state of a grammar with multiple goals corresponds to a start symbol inserted by Lily. This symbol has, as an alternative list, each goal specified in the parser specification of the multiple goal grammar. Thus interpreted, the single and multiple goal grammars may be treated homogeneously.

From a given state of the RTN, outward arcs are specified according to the director symbols of alternative derivations possible from the state. For terminal alternatives, an outward arc is labelled with a terminal symbol. For nonterminal symbols, action symbols, and the empty-string symbol, outward arcs are labelled with director symbols. In the case of nonterminal symbols, stack-pushing actions are included in the specification. Action symbols require the execution of user defined actions. For example, consider the following grammar:

```
[goal]: t1, t2 | @action, t3 | [n1] | #;
[n1]: t4 | t5 | t6;
```

This grammar will generate a parser conforming to an RTN like that shown in figure 4. Tables IIa and IIb show a representation of the two tables encoding the RTN. Note in the example that: t1...t6 are terminal symbols; @action is an action symbol; [goal], implicitly the grammar goal symbol, and [n1] are nonterminal symbols; and the empty string is represented by a pound sign.

4. Optimizing Tables: The Lily parser generator attempts to optimize parser function tables to decrease table size and improve parser speed. Small tables are desirable because they take up less space in the generated parser and

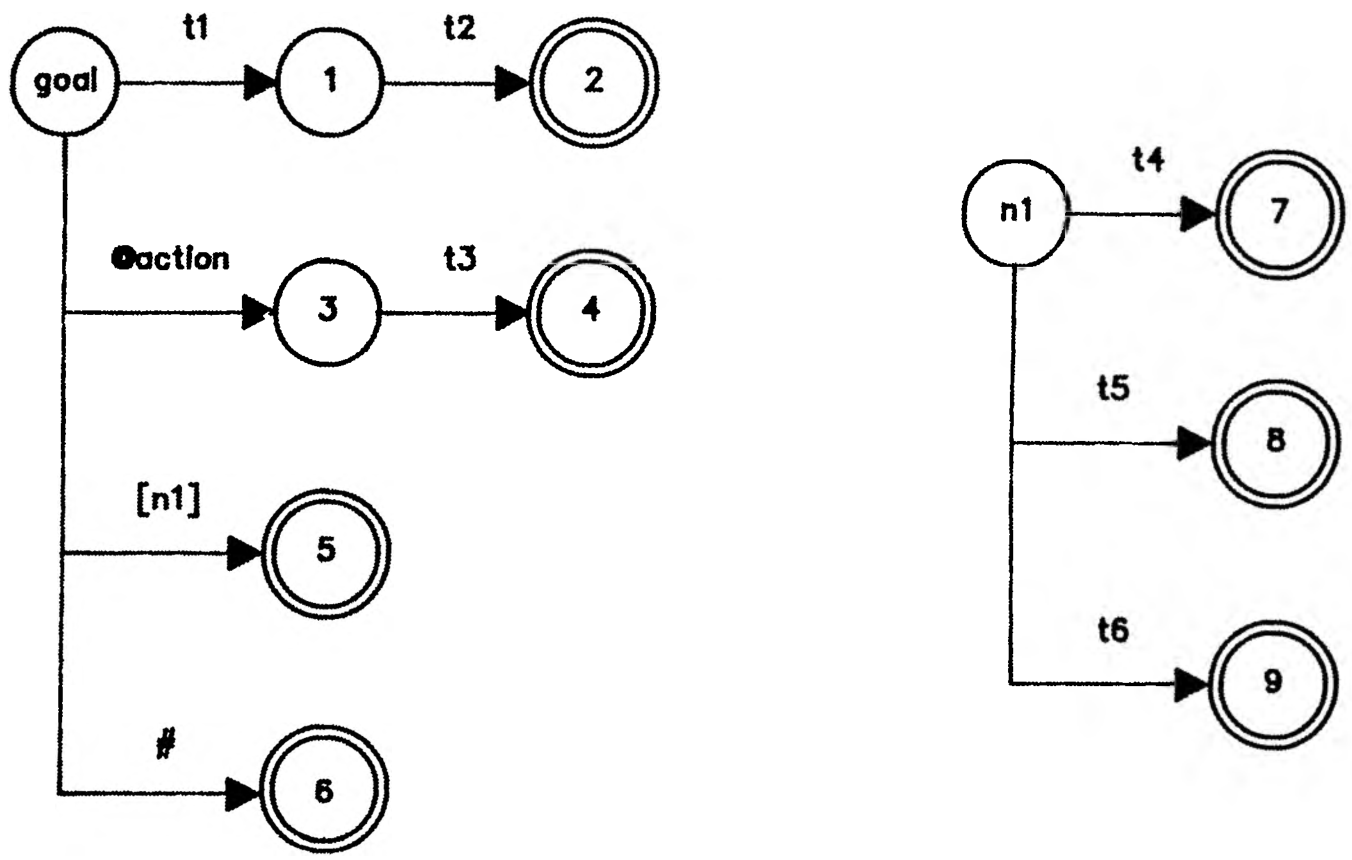

Figure 4. An RTN for the example grammar

TABLE II (a)

MAIN PARSING TABLE FOR EXAMPLE GRAMMAR

State	Token	Next	Action
goal (start)	t1	1	shift
goal	t3	3	user
goal	t4	n1	push 5
goal	t5	n1	push 5
goal	t6	n1	push 5
1	t2	2	shift
3	t3	4	shift

TABLE II (b)

DEFAULT TABLE FOR EXAMPLE GRAMMAR

State	Next	Action
goal	6	-
1	-	error
2	-	pop next
3	-	error
4	-	pop next
5	-	pop next
6	-	pop next
n1	-	error
7	-	pop next
8	-	pop next
9	-	pop next

because they require less effort on the part of the hash table generator programs. Execution speed in the generated parser is also important, computer time and user time being valuable resources.

Earlier in this paper it is mentioned that the repetitive enclosures in an RRPG, as opposed to their recursive counterparts, may be implemented in such a way as to reduce the stack-handling burden placed on the parser. Recursive specifications in linear grammars require the parser to execute stack pushes and pops for each instantiation of a recursive symbol. While this is necessary when a grammar contains structures which are inherently recursive (e.g. mathematical expression structures), the practice becomes wasteful when it is applied to structures which are iterative in nature (e.g. lists of identifiers for function parameters).

For this reason, Lily replaces the implicit recursive structures of repetitive closures with equivalent iterative structures. This is done by recognizing that stack pushes and pops in the recursive RTN occur at the beginning and ends of repetitive structures and serve to cancel one another out. Also, unnecessary intermediate states may sometimes be removed by suitable reconfigurations.

The method of reconfiguring a group of alternatives under transitive closure (i.e. between "(" and ")+") is illustrated by example. Consider the following Lily RRPG production using iteration and its recursive, linear counterpart:

```
[A]:    (a1 | a2 | ... | an)+; is equivalent to the
        following:
[A]:    [A'];
[A']:  a1, [A''] | a2, [A''] | ... | an, [A''];
[A'']:  [A'] | #;
```

Here, as before, a_i represents the ith alternative in a list of alternatives. Figure 5 shows the RTN for both the iterative and recursive versions. Note that state 1, corresponding to the recognition of an A' still remains in the iterative version while A'' is no longer necessary. The RTN, in the iterative case, resembles the finite automaton for a transitive closure.

Reconfiguring a set of alternatives enclosed between "(" and ")*" is accomplished in a similar manner. Consider the following iterative production and its recursive counterpart:

```
[A]:    (a1 | a2 | ... | an)*; is equivalent to the
        following:
[A]:    [A'] | #;
[A']:  a1, A' | a2, A' | ... | an, A';
```

Figure 6 depicts the respective recursive and iterative RTN's.

The most effective optimization, however, involves the selection of default alternatives. This is done by considering all of the possible arcs from a given state. Arcs labelled with nonterminals actually constitute more than one entry in the main parsing table; one entry is included for every director symbol of the nonterminal. A substantial amount of table space can be saved by simply making some nonterminal transitions by default.

This is accomplished by analyzing the possible alternative structures which may occur after a grammar symbol. In the Lily parser generator, an alternative may be chosen by default under three conditions:

1. No other alternative is nullable.
2. The structure is LL(1) (i.e. the director symbols of all alternatives are disjoint).
3. The alternative begins with a nonterminal.

Rule 1 applies because if a nullable alternative exists then that alternative will require the default transition from the current state. Rule 2 simply states that the parser must be deterministic. If this is not the case, then eliminating defaults allows the parser generator to determine the location of the ambiguity. Rule 3 is necessary because only nonterminals have corresponding states where the parser is "expecting to see" a given structure. Recall that states corresponding to all other symbols are entered only <u>after</u> seeing the symbol. If more than one nonterminal conforms to all three conditions, then the one with the largest set of director symbols is chosen. This has the effect of reducing the main parsing table size by the greatest possible amount.

An example serves to show how optimization is achieved using defaults. Consider the grammar given below:

```
[S]: [A] | [B] | t1;
[A]: t2 | t3 | t4;
[B]: t5 | t6;
```

Here, as above, t1, t2, ..., t6 represent arbitrary terminals.

In the alternative list comprising the right side of [S], both nonterminals, [A] and [B], meet the three criteria given above. However, the director symbols of [A] (t2, t3, and t4) outnumber the director symbols of [B] (t5 and t6). In this case [A] will be chosen as the default nonterminal. The result of this is that whenever the parser is in the state corresponding to [S], a token other than t1, t5 or t6 will result in a transition to a state where the parser will expect to see an [A]. Tables IIIa and IIIb show the parser tables before applying the default-selection optimization while tables IVa and IVb show the tables after application.

In practice, nonterminals will often be the only alternative which may occur after a given symbol. These nonterminals will be chosen as the defaults automatically. For example, consider the following production:

```
[A]: t1, [B];
```

Here, after seeing a t1 token, the parser will automatically expect to see a [B], regardless of the token currently being scanned.

TABLE III (a)

MAIN PARSING TABLE BEFORE OPTIMIZATION

State	Token	Next	Action
S	t1	1	shift
S	t2	A	push 2
S	t3	A	push 2
S	t4	A	push 2
S	t5	B	push 3
S	t6	B	push 3
A	t2	4	shift
A	t3	5	shift
A	t4	6	shift
B	t5	7	shift
B	t6	8	shift

TABLE III (b)

DEFAULT TABLE BEFORE OPTIMIZATION

State	Next	Action
S	-	error
A	-	error
B	-	error
1	-	pop next
2	-	pop next
.	.	.
.	.	.
.	.	.
8	-	pop next

TABLE IV (a)

MAIN PARSING TABLE AFTER OPTIMIZATION

State	Token	Next	Action
S	t1	1	shift
S	t5	B	push 3
S	t6	B	push 3
A	t2	4	shift
A	t3	5	shift
A	t4	6	shift
B	t5	7	shift
B	t6	8	shift

TABLE IV (b)

DEFAULT TABLE AFTER OPTIMIZATION

State	Next	Action
S	A	push 2
A	-	error
B	-	error
1	-	pop next
2	-	pop next
.	.	.
.	.	.
.	.	.
8	-	pop next

III. CONCLUSION AND SUGGESTIONS FOR FURTHER WORK

The Lily parser generation system should provide a useful tool for generating compiler front ends. Much effort has gone into the design and implementation of the system in order to make it both easy to use and relatively powerful. Although extensive validation of the system has been carried out, further use of the system will lend assurance to the system's correctness and practicality.

Many opportunities exist for expansion. These range from trivial extensions of convenience to large-scale improvements in power. In this chapter suggestions are offered pertaining to future augmentations of the system. It is hoped that these suggestions will prove helpful to anyone considering such a project.

A. ADDING ATTRIBUTE TRANSLATION

Retooling Lily to allow for the formal translation of attributes would undoubtedly be a very difficult task. Indeed, the effort required for such an undertaking would, in all probability, considerably exceed the total effort invested in creating the current prototype. Notwithstanding this, formal attribute evaluation would significantly improve Lily, possibly making it a market-viable software product.

A good model for an attribute-processing compiler-compiler exists in the Visible Attribute Translation System (VATS) (in Tremblay and Sorenson, cited in chapter II).

Lily already resembles VATS in a number of ways, particularly in the way semantic actions are handled; i.e., both systems use action symbols. In VATS the user may specify attributes to be associated with grammar symbols. It is then the user's responsibility to see that, through actions, the attributes are evaluated in the correct order.

Since Lily uses RRPG's, traditional attribute grammar schemes will prove insufficient. This results from the implementation of repetitive closures to express iterative structures. The problem is: what should be done with the attributes of nonterminals within these closures? It is, for instance, clear that an iterated nonterminal should be able to inherit attributes from previous iterations (siblings in the parse tree). However, linear attribute grammars, by nature, cannot provide a solution to this problem.

Jullig and DeRemer [19] propose methods for dealing with Regular Right Part Attribute Grammars (RRPAGS). These include: list creation, in which iterations of nonterminals create a list of corresponding attributes; list distribution, in which attributes of iterated nonterminals are inherited from a corresponding list element; and a "bucket brigade," where attributes are passed from iteration to iteration.

Apart from these methods, most attributes can be handled in one of two ways. First, a Lily generated parser could be made to create and support an attributed parse tree as in LINGUIST-86. Table-driven methods could still be used in this case. A second method would be much easier from the standpoint of run-time attribute support. This involves the abandonment of the table-driven approach and switching to recursive descent in the generated parsers. Most attributes could then be passed as parameters between the implied mutually recursive functions.

B. IMPROVEMENTS TO PRODUCTION EXPRESSIONS

Less ambitious than formal attribute translation would be the augmentation of Lily production expressions. Changes can be made which would make grammar definition more convenient, powerful, and conservative of space.

One possibility is the addition of hexadecimal integer constants. These would be useful since some Lily parser functions receive input corresponding to bytes. The current implementation forces the user to describe these bytes either as ASCII characters or as decimal integer constants. Hexadecimal integers are sometimes more natural for the expression of byte integers. This would be particularly clear if a user, for example, wished to use Lily to generate a small unassembler.

Another good possibility is the inclusion of further regular right part constructs. LaLonde (cited in chapter II) mentions two alternatives which, incorporated into Lily, might take the following forms:

1. (p)? = # | (p)

2. (p) %list (q) = (p), ((q), (p))*

where p and q are Lily production expressions.

The first extension specifies that p is an optional structure in the grammar. The second, somewhat more complicated, defines a list structure, in this case a list of p's separated by q's. Such an expression could be used, for instance, to describe a list of identifiers separated by commas.

C. AUTOMATIC DISAMBIGUATION

Certain non-LL(1) grammatic structures should be susceptible to automatic disambiguation. Improving Lily to perform this task would greatly increase the system's convenience to the user. Lexical analyzer tokens, for example, might be able to have common prefixes without necessitating disambiguation by the user.

A method worth considering is the application of standard algorithms for the conversion of nondeterministic finite automata (NFA) to equivalent deterministic finite automata (DFA). One such algorithm, which may be found in Hopcroft and Ullman [20], converts NFA's to equivalent DFA's by creating compound states whenever nondeterminism occurs. This algorithm could be modified to attempt disambiguation of an RTN. Since nonterminals, action symbols, and sets are not included in NFA's, the algorithm could be constructed to give up if one of these symbols is encountered.

An example of a grammar which might be automatically disambiguated is:

```
     [colon]:       ':';
     [assign]:      ':', '=';
     [produces]:    ':', ':', =;
```

Here it is assumed that the three left-side nonterminals are goals of a multiple goal grammar. Thus, they are implicitly the alternatives of a single, Lily-supplied goal symbol. Figure 7 shows an NFA for this grammar followed by the corresponding DFA.

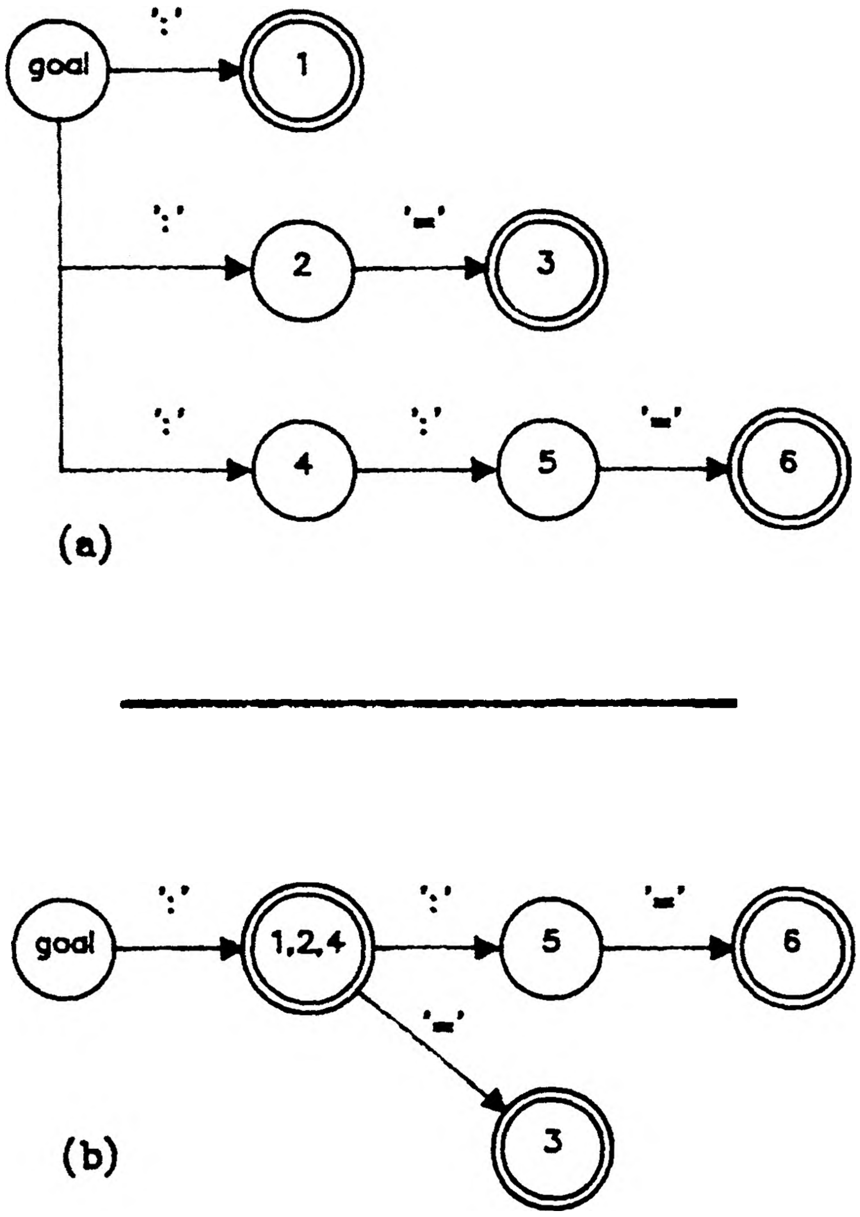

Figure 7. (a) NFA for example grammar
(b) Equivalent DFA

BIBLIOGRAPHY

1. Sager, Thomas J., "Lily--A Generator for Parser Frontends," currently unpublished manuscript.

2. Lesk, M. E., and E. Schmidt, "Lex--A Lexical Analyzer Generator," UNIX Programmer's Manual Supplementary Documents, University of California-Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

3. Johnson, S. C., "YACC: Yet Another Compiler-Compiler," UNIX Programmer's Manual Supplementary Documents, University of California-Berkeley Software Distribution, Virtual VAX-11 Version, March 1984.

4. Lewi, J., et al., A Programming Methodology in Compiler Construction, North-Holland, Amsterdam-New York-Oxford, 1979.

5. Kastens, U., B. Hutt, and E. Zimmerman, GAG: A Practical Compiler Generator, Springer-Verlag, Berlin-Heidelberg-New York, 1982.

6. Knuth, D. E., "Semantics of Context-free Languages," Math. Systems Theory, Vol. 2, June 1968, pp. 127-145.

7. Farrow, R., "Generating a Production Compiler from an Attribute Grammar," IEEE Software, October 84, pp. 77-93.

8. Ganapathi, M., C. N. Fischer, and J. L. Hennessy, "Retargetable Compiler Code Generation," Computing Surveys, Vol. 14, No. 4, December 1982, pp. 573-592.

9. Leverett, B. W., et al. "An Overview of the Production-Quality Compiler-Compiler Project," IEEE Computer, Vol. 13, No. 8, August 1980, pp. 38-49.

10. Ganzinger, H. K., K. Ripken, and R. Wilhelm, "Automatic Generation of Optimizing Multipass Compilers," Proceedings IFIP Congress 77, 1977.

11. Dewdney, A. K., "Of Fractal Mountains, Graftal Plants, and Other Computer Graphics at Pixar," Scientific American, December 1986, pp. 14-20.

12. LaLonde, W. R., "Regular Right Part Grammars and Their Parsers," Communications of the ACM, Vol. 20, No. 10, October 1977, pp. 731-741 (p. 731 quoted).

13. Sager, Thomas J., "A Polynomial Time Generator for Minimal Perfect Hash Functions," Communications of the ACM, Vol. 28, No. 5, May 1985, pp. 523-532.

14. Wirth, Niklaus, "From Programming Language Design to Computer Construction," Communications of the ACM, Vol. 28, No. 2, February 1985, pp. 160-164.

15. Knuth, D. E., "Top-Down Syntax Analysis," Acta Informatica 1, 1971, pp. 79-110.

16. Tremblay, J. P., and P. G. Sorenson, The Theory and Practice of Compiler Writing, McGraw-Hill Book, New York, 1985, pp. 208-273, pp. 722-764.

17. Griffiths, M., "LL(1) Grammars and Analyses," Lecture Notes in Computer Science--Compiler Construction, No. 21, Springer-Verlag, New York, 1974, pp. 57-84.

18. Woods, W. A., Transition Network Grammars for Natural Language Analysis, Communications of the ACM, Vol. 13, No. 10, October 1970, pp. 591-606.

19. Jullig, R., and F. DeRemer, "Regular Right-Part Attribute Grammars," Proceedings of the SIGPLAN Symposium on Compiler Construction, June 1984.

20. Hopcroft, John E., and Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, Mass., 1985.

VITA

Timothy Topper Taylor was born on May 18, 1961 in Peoria, Illinois. He received his primary education in: Joplin, Missouri; Columbia, Missouri; Salt Lake City, Utah; Los Alamitos, California; and Chillicothe, Illinois. He received his secondary education in Kirksville, Missouri. He received his Bachelor of Science Degree in Mathematics from Northeast Missouri State University, Kirksville, Missouri, in May 1985.

Tim has been enrolled in the Graduate School of the University of Missouri-Rolla, in Rolla, Missouri, since August 1985.

APPENDIX A

SYNTAX FOR THE LILY PARSER GENERATION LANGUAGE

The following grammar uses an extension of BNF (Backus-Naur Form). Braces, "{" and "}" indicate that the enclosed structure is to be repeated zero or more times. Brackets, "[" and "]", indicate an optional structure. The pound sign, "#", indicates the empty_string. The sheffer stroke, "|" indicates alternation while concatenation is symbolized by the null operator. Concatenation takes precedence over alternation. Underlined items indicate tokens while identifiers without underlining indicate nonterminals.

```
front-end::=
     global-part {parser-part} %end ;

global-part::=
     [%global pascal-code]

pascal-code::=
     $$  (a block of Pascal text) $$

parser-part::=
     %parser standard-name [%look integer] ; |
     %parser identifier [att-decl] [%look integer] ;
       parser-tail

standard-name::=
     identifier

att-decl::=
     ( att-list-decl {; att-list-decl} )

att-list-decl::=
     att-ident-list : type-ident

att-ident-list::=
     attribute {, attribute}

attribute::=
     inh-att-ident | syn-att-ident
```

```
type-ident::=
     identifier

parser-tail::=
     receives-part [%local pascal-stuff] goals-part
     [keywords-part] [sets-part] [%fore pascal-code]
     productions-part [null-stmt] [%aft pascal-code]
     [actions-part] [disambig-part]

receives-part::=
     %receives parser-name [rec-att-list] ;

parser-name::=
     identifier

rec-att-list]::=
     ( identifier {, identifier} )

goals-part::=
     %goals goal-ident {, goal-ident} ;

goal-ident::=
     identifier | ( identifier )

keywords-part::=
     %keywords : type-ident = keyword-list ;

type-ident::=
     identifier

keyword-list::=
     identifier {, identifier}

sets-part::=
     %sets set-definition ; {set-definition ;}

set-definition::=
     identifier = set-expression

set-expression::=
     set-term {addop set-term}

addop::=
     + | -

set-term::=
     set-factor {* set-factor}

set-factor::=
     ordinal-range | - set-factor | identifier | ( set-
     expression )

ordinal-range::=
     ordinal [.. ordinal]
```

```
ordinal::=
     character | integer

productions-part::=
     %productions production ; {production ;}

production::=
     nonterm-ident : automaton

automaton::=
     auto-term {| auto-term}

auto-term::=
     auto-factor {, auto-factor}

auto-factor::=
     character | integer | identifier | nonterm-ident |
     # | ( automaton closure

closure::=
     ) | )+ | )*

null-stmt::=
     %null : automaton ;

actions-part::=
     %actions identifier pascal-code {identifier
     pascal-code}

disambig-part::=
     %disambig triple pascal-code {triple pascal-code}

triple::=
     integer , integer , integer | integer , # ,
     integer
```

APPENDIX B
A MANUAL FOR LILY

INTRODUCTION

This manual is divided into two chapters. The first chapter deals with the Lily language and discusses the structure and meaning of the various parts in a Lily program. Following this is a tutorial-style chapter on using the Lily system programs and writing Lily source code. Tested examples of Lily source code are given along with advice which might be of use to those who wish to utilize the system.

Both this manual and the Lily system (object code only) are released to the public domain and may be freely copied and distributed provided that such actions are not undertaken for purposes of financial gain. Lily originates at the University of Missouri-Rolla Department of Computer Science and further information about the system may be obtained from Dr. Thomas J. Sager of that department.

I. <u>THE LILY LANGUAGE</u>

A Lily program, diagrammed in figure 8, consists of an optional global part and a series of zero or more parser specifications, followed by the key word %END. From this source program the parser generator (metacompiler hereafter) produces parsers for LL(1) languages, a reasonably powerful subset of the deterministic context-free languages. In

```
%global  (optional)
$$
.
.        (Pascal code)
.
$$

%parser ...                ---------------------+
                                                |
  %receives ...                                 |
                                                |
  %local   (optional)                           |
  $$                                            |
  .                                             |
  .        (Pascal code)                        |
  .                                             |
  $$                                            |
                                                |
  %goals ...                                    |
                                                |
  %keywords ...  (optional)                     |
                                             ___|___
  %sets ...      (optional)
                                        (zero or more of)
  %fore          (optional)
  $$                                         _______
  .                                             |
  .        (Pascal code)                        |
  .                                             |
  $$                                            |
                                                |
  %productions ...                              |
                                                |
  %null ...   (optional)                        |
                                                |
  %aft   (optional)                             |
  $$                                            |
  .                                             |
  .        (Pascal code)                        |
  .                                             |
  $$                                            |
                                                |
  %actions ... (optional)                       |
  %disambig ... (optional) ---------------------+

%end
```

Figure 8. Diagram of a Lily source module

addition, using features for reconciling prefix ambiguities of finite length, parsers for LL(k) languages may be generated. This chapter defines the structure and meaning of each of Lily's component parts as they occur in order within a Lily source file. It may be helpful to refer to figure 8 throughout this chapter.

The terminal symbols (tokens) of the language are depicted in table V. They consist of key words (listed in table VI), synthesized and inherited attribute identifiers, terminal symbol identifiers, nonterminal symbol identifiers, action symbol identifiers, and various operators and separators. Identifiers which are equivalent to terminal symbol identifiers are also used for defining parser names, set names and various other source language particulars. When an identifier is used in one of these latter contexts it is simply referred to as an identifier. Comments, enclosed between braces, may be included anywhere in the source code that a token may occur and are ignored by the parser generator (i.e. {This is a Lily comment.}).

Letter cases in identifiers and key words are insignificant and are converted to uppercase in the course of metacompilation. Thus the key word identifiers "%GloBaL" and "%global" are equivalent as are identifiers "ParserX" and "parserx".

Blocks of Turbo Pascal text, used in the global, local, fore, aft, actions, and disambig parts of a Lily program, are placed between pairs of dollar signs ("$$") and will be

TABLE V

TERMINAL SYMBOLS OF LILY LANGUAGE

Token		Meaning
identifier	--	Terminal symbols and all-purpose
[identifier]	--	Nonterminal
<identifier	--	Synthesized attribute
>identifier	--	Inherited attribute
@identifier	--	Action symbol
%identifier	--	Key word
number	--	Integer constants from 0 to 32767
(	--	Begins parenthetic enclosure
)*	--	Ends parenthetic enclosure (Kleene closure)
)+	--	Ends parenthetic enclosure (transitive closure)
)	--	Ends parenthetic enclosure (simple operator override)
:	--	Colon (separator)
;	--	Semicolon (separator)
|	--	Sheffer stroke (alternative operator)
,	--	Comma (separator and concatenative operator)
..	--	Two periods (separator in subranges)
=	--	Equal sign (separator and set assignment operator)
'c'	--	ASCII character (c arbitrary)
*	--	Asterisk (set intersection operator)
+	--	Plus sign (set union operator)
-	--	Minus sign (Unary absolute complement, relative set complement operator)
#	--	Pound sign (null string constant)
$$	--	Dollar sign pair (enclosure for Pascal code blocks)

TABLE VI

KEY WORDS OF LILY LANGUAGE

Key word		Meaning
global	--	Precedes global definitions and declarations
parser	--	Begins a parser specification
look	--	Used to request look-ahead
goals	--	Precedes a list of goal symbols
actions	--	Precedes action definitions
receives	--	Precedes specification of received parser
keywords	--	Precedes a list of key words
sets	--	Precedes set definitions
productions	--	Signals beginning of productions
null	--	Used to specify a right side to be ignored
end	--	Signals the end of Lily source
disambig	--	Begins a list of parser overriding actions
local	--	Precedes local declarations
fore	--	Precedes an action done prior to parsing
aft	--	Precedes an action done after parsing

referred to as <u>Pascal code blocks</u>. These blocks are copied verbatim by the metacompiler and are not checked for proper Pascal syntax or semantics. Naturally, the use of dollar sign pairs as delimiters implies that any Pascal code block used in a Lily source program must not contain a pair of dollar signs. Furthermore, a comment which begins outside a Pascal code block may be terminated by a right brace inside the block with a syntax error the probable result. For this reason comments within Pascal code blocks are best enclosed within the Turbo Pascal comment delimiters "(*" and "*)".

Certain standard identifiers are used in the parsing functions generated by Lily and these may be accessed in Pascal code blocks to affect the parsing sequence. An alphabetical list containing each standard identifier along with its class (i.e. constant,variable, or procedure) is given in table VII. Note that all standard constants and variables are of type integer.

The constant "accept_" is meant to be used in an action. Making the assignment "state_:= accept_" in an action causes the parser to exit.

The variable "action_" contains the number of the current action to be executed by the parser. This variable can be used to cause the parser to shift by assigning "action_:= shift_" in a Pascal code block. Most often the variable will be used in ambiguity-resolving actions.

The constant "error_" can be used in an action to indicate that the return value of the parser is an error.

TABLE VII

STANDARD IDENTIFIERS IN LILY PARSERS

Identifier	Class
accept_	constant
action_	variable
error_	constant
no_action_	constant
pop_	procedure
push_	procedure
return_	variable
shift_	constant
start_state_	constant
state_	variable
token_	variable

This is done using the "return_" variable discussed below.

The action constant "no_action_" is a value which can be taken on by the variable "action_" and designates to the parser that no action is to be taken.

Procedures "pop_" and "push_" are available only in parsers which require the use of a stack (namely, those parsers containing nonterminal references in production right sides). Under normal circumstances the Lily system user need not worry about using these procedures as Lily handles all stack operations automatically. However, when a parser specification contains ambiguities it is sometimes necessary to override the operation of the generated parser and take control of the stack in an ambiguity-resolving action.

The "pop_" procedure sets the current state of the parser, held in the variable "state_" to the value on the top of the stack. Then the top of the stack is discarded and replaced with the next element down in the stack. If the stack is empty when it is popped, the state of the parser is set to the constant "accept_."

The "push_" procedure is the logical complement of the "pop_" procedure and is used to push its single parameter, an integer state value, onto the stack. If the stack is full (one hundred elements is the current default value allowed), then an error message is output while the parser is permitted to continue operating.

The "return_" variable holds the integer value to be returned by the parser. It is set, either automatically or in a user action, prior to exiting the parser function.

The "shift_" constant can be assigned to the action variable "action_" causing the parser to execute a shift. This is normally done for purposes of ambiguity resolution.

The "start_state_" contains the value for the state in which the parser is to start. At the beginning of each parser call the state variable "state_" of the parser is set to the constant "start_state_." The user may also make such an assignment in his actions part. Note however, that the starting token value is not reset.

The variable "state_" contains the integer state of the current parser. Assignments made to this variable in user actions cause the parser to change states.

The variable "token_" contains the value of the current token being processed by the parser. However, it is merely a copy of the parsers working token. As such it is intended only to be examined by user actions and not modified. Modifying this variable will have no effect on the parse sequence.

Finally, the working token of the parser, unlike "token_" affects the action of the parser. The variable is global so that it will remain unchanged from the end of one parser call to the beginning of the next. The name of the working token variable for a given parser is formed by prefixing the first eight characters of the parser's name to the string "_token_". Thus for the parser "lex_anal" the working token may be referenced as "lex_anal_token_". The practice of altering the working token is not recommended although the possibility has been mentioned here for the sake of completeness.

A. GLOBAL PART

The optional global part, which consists of the key word %GLOBAL followed by a Pascal code block, is used to declare variables and to define types, functions, and procedures that are to be considered global to the generated parser functions. For example consider the following:

```
%global
$$
type buffer_type = string[20];
var buffer: buffer_type;
procedure x;
  begin
  { ... }
  end;
$$
```

In this example the user has defined a type, declared a variable, and defined a procedure as global to the parsers that will follow. Access to these definitions and declarations will be available to any Pascal code block in any subsequent parser specification.

B. PARSER HEADING

The key word %PARSER begins a Lily parser specification. The definition of a parser in this specification continues until the next %PARSER key word or the %END key word is reached. Within a parser specification the parser being defined is referred to as the _current_ parser.

Following the key word %PARSER is an identifier naming the parser, then an optional parenthesized declaration of attributes, and finally an optional request for look-ahead buffering. This sequence is terminated by a semicolon. The name of the parser is an identifier which is significant only in the first eight character positions. This situation arises from the fact that DOS (Disk Operating System) files are created by the Lily system and these are given names which correspond to parser names.

Attribute declarations take a form similar to the formal parameter declarations of Pascal procedures and functions. Indeed, this resemblance is directly attributable to the fact that in the final phase of metacompilation, the attributes of a particular parser are used to define the formal parameters in the generated parsing function. At metacompile time synthesized attributes of a parser are declared to be pass-by-reference (Pascal VAR) and are therefore available to and modifiable by statements in that parser's Pascal code blocks. Inherited attributes are declared pass-by-value and are available but are not modified upon returning from the parser function.

Parser attributes are declared in a list of one or more attribute declaration instances separated by semicolons. Each attribute declaration instance consists of a list of attribute identifiers followed by a colon, and ended by an identifier. The ending identifier should be a one-word

Pascal type, either standard or user-defined, which is global to the current parser. If the identifier represents a user-defined type then it is preferable for reasons of consistency that it be defined in the Lily global part but it may also be defined in an available scope of the user's driver program.

Look-ahead buffering for the parser, if any is needed, is requested after the optional attribute declarations. The key word %LOOK is used followed by an integer constant representing the maximum amount of look-ahead required. This is useful when the parser receiving its tokens from the current parser needs to see more than one token ahead in order, perhaps, to resolve ambiguities.

When look-ahead buffering is specified the user is given access to an automatically generated look-ahead function. This function will have a name formed by prefixing the string 'look_' to the current parser's name. The look-ahead function has all of the attributes of the current parser as well as one more to indicate how far look-ahead is to be carried out on a given call.

The following are examples of a parser headings:

```
%parser lex;

%parser lex(>inbuffer, <outbuffer: buffer_type;
  <buffer_length: integer);

%parser lex(>insomething: some_type;
  <value: integer) %look 2;
```

In the first example a parser to be named "lex" is specified. It has no attributes and no look-ahead function will be generated for it. The second example shows a parser with three attributes, one inherited and two synthesized. The first two attributes have type "buffer_type" while the last attribute has type integer. In the third example a parser requires two attributes, the first inherited and the other synthesized. Further, the generation of a look-ahead function capable of scanning ahead a maximum of two tokens is requested. Since the name of the parser, as in the first two examples, is "lex," the user could access the generated Pascal look-ahead function by, for instance, assigning

```
x:= look_lex(dist, val_to_send, val_to_receive);
```

This means that we want to assign "x" the integer value of the token which is "dist" tokens beyond the current token in the input. Also we want the inherited attribute called "val_to_send" to be used while determining the look-ahead token value and we want to receive "val_to_receive" after it has been synthesized.

Apart from user-defined parsers, Lily also provides <u>standard</u> parsers. These give the system user a convenient, automatic way of implementing low-level input to the parsers he or she defines. Two versions with slightly differing capabilities are provided named "GETCHAR" and "GETCHAR_L" respectively. Both parsers read characters one-by-one from a standard file and return corresponding ASCII codes as integers. The difference between the two standard parsers is that "GETCHAR_L" automatically outputs a line-numbered listing to a standard listing file.

The standard file read by the standard parsers is called "input_file" while the standard file used for listings is called "list_file." Whenever a standard parser is used in a Lily program one or both of these files must be assigned to user-declared file variables according to the conventions of Turbo Pascal. In addition, the file used for input must be reset while the file used for listing output must be rewritten.

These two files are to be distinguished from the standard input and output files of Pascal. A user wishing to utilize the standard files (perhaps to allow his program to serve as a filter in a pipe) must explicitly assign the Lily standard names to these files. Other considerations, such as defining the buffer size for the Pascal standard files, are also left up to the user.

The standard parsers differ from user-defined parsers in the way they are included in the Lily source. Instead of defining a full parser specification for standard parsers the user includes only the parser heading in his definition. In addition, while the inclusion of a look-ahead request is allowed for standard parsers, there must be no attribute declaration list present. For example consider the following parser headings:

```
%parser getchar;
%parser getchar_l;
%parser getchar_l %look 2;
```

In the first example the parser GETCHAR is made available. No listing file will be generated at compile time (as differentiated from metacompile time). In the second example parser GETCHAR_L is made available. It functions in exactly the same way as GETCHAR but a listing will be generated to a standard listing file at compile time. The last example shows a standard parser that generates a listing and also allows look-ahead under the same terms as the look-ahead described for user-defined parsers.

C. RECEIVES PART

The receives part of a parser specification is used to give Lily information about the current parser's source for tokens. The key word %RECEIVES is followed by an identifier, an optional parenthesized list of identifiers separated by commas, and a terminating semicolon. The identifier gives the name of a lower level parser which will be called upon to supply tokens during the operation of the current parser. This received parser can be either a previously specified Lily parser (standard or user-defined) or, less conveniently but legally, the function name of a subprogram which the user supplies.

The optional parenthesized identifier list contains the names of the actual parameters to be used when calling the received parser function. Such a call is made whenever the current parser executes a shift action. If the actual parameter list facility is used it should be used with care and only with a complete understanding of the functioning of Lily-generated parsers. The parameter names in the list are not declared by Lily in the generated Pascal function and must be declared by the user in the local part of the current parser, in the global part of the Lily source, or in the user's driver program. If the current parser receives a non-Lily, user-supplied parser, then the omission of the parameter list indicates that the received parser has no attributes.

On the other hand, if the current parser receives from a parser previously defined in the Lily source, then the user need not consider the parameter list at all. Lily will copy the names from the attribute list of the received parser and make sure that copies local to the current parser are available to the current parser's Pascal code blocks. For example, assume parser X with attributes >a1, <a2, <a3 is received by parser Y. Then leaving off the parameter list in the receives part of parser Y tells Lily to make Y-local copies of the attributes of X. Thus, the Pascal code blocks of parser Y may make use of variables a1, a2, and a3. Whenever the received parser, X, is called for a token by Y, the attributes of X will initially be set to the values

assigned to them in the Pascal code blocks of Y. Such an assignment will be necessary only for the inherited attributes of X.

Nevertheless a parameter list in the receives part can still be specified if the user wishes to call the local copies of the received parser's attributes by names which differ from those of the attributes in the received parser. As above, all parameter names used in the parameter list will have to be declared by the user.

In most cases it will be best for the current parser always to receive Lily-defined parsers since this allows for easy, automatic interfacing of parser functions. While support for receiving non-Lily parsers exists, the use of that support introduces minor complexities as well as certain necessities of detail which the user may find tedious. Since the purpose of the language is to provide convenience, the support for non-Lily received parsers constitutes a slight digression from the design philosophy of the language. The inclusion of these facilities is motivated by a desire for generality and versatility.

Notwithstanding this disclaimer, the use of a non-Lily received parser has two effects on the current parser. First, identifier names of the received parser (such as its goals and key words) will be unavailable for use in the current parser's productions and sets. This means that the sets and productions of the current parser must use either integer or character ordinals to refer to the tokens of the received parser. Second, the set universe of the current parser is assumed to be {x | x is an integer element of [0, 255]}. In other words, Lily assumes that the tokens coming from a non-Lily received parser will range from 0 to 255.

The following are examples of the receives part:

```
%receives getchar_1;
%receives Lily_lex;
%receives NonLily_lex( a, b, c);
```

In the first example the current parser will receive its tokens from the standard parser getchar_1 which was presumably defined in a preceding parser heading. The second example shows the current parser receiving its tokens from Lily_lex, a user-defined parser supposed defined in an earlier parser specification. Pascal code blocks of the current parser can in this case make use of local variables whose names are identical to the attribute names used in Lily_lex.

The final example shows a current parser being defined as receiving its tokens from a source function which is assumed <u>not</u> to have been defined in a preceding parser specification. In this case the user will supply a function in his driver program to supply the correct integer tokens. Lily will use the parameters "a, b, and c" when it needs to call the user's function to execute a shift. These three parameters should be declared or defined by the user in the global part, in the current parsers local part or in the driving program.

D. LOCAL PART

The local part in a parser specification is used like the global part of a full Lily program. Its purpose is to supply the system user with the ability to declare Pascal variables as well as to define types and subprograms. These definitions and declarations are made local to the function generated by the metacompiler as it implements the current parser. Consequently, they are accessible only to those Pascal code blocks which are within the current parser.

The local part, which is optional, consists of the key word %LOCAL followed by a Pascal code block containing definitions and declarations. For example, consider the following local part.

```
%local
$$
type byte_array = array [0..10] of byte;
var byte_table: byte_array;
procedure x;
  begin
  { ... }
  end;
$$
```

In this example, the user defines a type, declares a variable of that type, and then defines a procedure. These declarations and definitions will be local to the function generated for the current parser.

E. GOALS PART

The goals part, required in each user-defined parser specification, consists of the key word %GOALS followed by a list of one or more identifiers any of which can be enclosed in parentheses. Identifiers, parenthesized or not, are separated by commas while the list of identifiers is terminated by a semicolon.

In the goals part of a parser specification the system user lists identifiers corresponding to nonterminal symbol identifiers in the left sides of productions within the current parser specification. These productions are then considered to define goal symbols for the current parser. If more than one goal is specified in the goal list, the current parser is referred to as a <u>multi-goal</u> parser hereafter in this manual and its grammar is called a multi-goal grammar. If the goal list contains only one goal, then the current parser is referred to as a <u>single-goal</u> parser while its grammar is called a single-goal grammar.

Each goal can be defined as being in one of two modes depending on whether or not the corresponding identifier is parenthesized in the goal list. If the identifier is not parenthesized, then the goal is in default mode. A parenthesized identifier indicates that the goal is in non-default mode.

A goal in default mode causes the current parser to return a unique integer on recognition of the goal's right side. In contrast, a goal in non-default mode can be recognized by the current parser but no provision is made for returning a corresponding integer and the return value defaults to an error signal. This means that the system user must provide an action at the end of the non-default goal's right side in order to designate a value to be returned.

For convenience Lily does in fact assign a unique token value to a non-default goal but provides no automatic way for the current parser to return that value. Instead, Lily will declare the name of the non-default goal as an integer constant in the function generated for the current parser. This constant is assigned the goal's token value and is available to the Pascal code blocks of the current parser. This can prove useful in the Lily code for equipping a parser with the ability to recognize key words.

The purpose of non-default goals is to facilitate the recognition of user-language key words (reserved words of the language for which a Lily front end is being specified). To allow the current parser to recognize key words a goal for recognizing identifiers is listed as non-default in the goal list and defined in a production. The production specifies the structure of an identifier and includes actions for creating a buffer of the identifier recognized. Then in an action at the end of the production, a return value for the goal is designated by calling the Lily-generated "search_key" function. This function checks the buffer against a list of key words specified by the user in the key words part of the current parser specification. A description of the use of the key word searching function will follow in the next section's discussion of the key words part. There, the Lily implementation of key words is considered in more detail.

Examples of the goals part follow:

```
%goals
  left_paren, right_paren, asterisk, (ident);

%goals
  program;
```

In the first example the current parser is defined to be a multi-goal parser, perhaps a lexical analyzer. Each of the goals in the list must be defined in the productions part of the current parser. Unique token values will be returned automatically by the parser function for all of the listed goals except "(ident)" the return value for which must be stipulated in a user action. A constant containing a token number for "ident" will be included in the current parsers Lily-generated Pascal function. This constant can be used or discarded as need be.

The second example shows the current parser being defined as a single-goal parser. This goal must be defined in the productions part of the current parser. If the parser function succeeds in recognizing a structure corresponding to this goal then it will return an integer zero. Otherwise, an error value is returned.

F. KEYWORDS PART

The keywords part consists of the key word %KEYWORDS, a colon, an identifier, an equals sign, and a list of identifiers terminated by a semicolon. The identifier following the colon is the type to be used for all of the current parser's reserved words. This type is a one-word identifier which should be defined in the local part of the current parser, the global part of the current Lily program, or in the user's driver program. For compatibility, the type must be a string of some arbitrary length.

Following the equals sign is a list of identifiers defining the key words of the user's language. The identifiers in the list, which must be at least one in number, are separated by commas. Since Lily is not sensitive to the cases of letters in the identifiers it cannot support case sensitivity in the key words of user defined parsers. All of the key words listed in the current parser's key word list are converted to uppercase just like other identifiers in the source. Thus, all keywords in the language being defined using Lily must be either uppercase or caseless as in Pascal.

Examples of the keywords part follow. Note that both examples are equivalent since all of the identifiers are converted to uppercase by Lily.

```
%keywords: some_type = if, then, for, do, while;
%keywords: some_type = If, Then, For, Do, While;
```

When the keywords part is used, the metacompiler generates a standard word searching integer function called "search_key" which is made local to the function for the current parser. The search_key function, available to all of the Pascal code blocks in the current parser, is called from an action at the end of a non-default goal production (See "GOALS PART" above). The purpose of the search_key function is to determine if a buffer, supplied in a parameter of the function matches any of the key words defined for the current parser. If a match is found, the token value corresponding to the matched key word is returned. Otherwise a default value specified by the user is returned.

The search_key function takes two actual parameters. The first parameter is a buffer which must be of the type given after the colon in the key words part. This buffer should be declared by the user either in the local part or as an attribute of the current parser. The second parameter is an integer indicating the default value to be returned by the search_key function if the search fails. Ordinarily this parameter will be a constant but if it is not then it should be declared as in the case of the first parameter.

Please refer to figure 9 for an example fragment of Lily code showing the implementation of key words in a current parser. The figure shows the definition of a parser ("example_parser") which receives its tokens from the standard parser GETCHAR. A conjunction of Lily parts in the example parser's parser specification serves to allow the parser to recognize key words and return unique integer

```
%global
$$
type buffer_type = string[20];
$$
                  .
                  . {other parser specifications}
                  .
%parser example_parser(<buf: buffer_type);
%receives getchar;
%goals goal1, goal2, (ident), goal4 { ... } ;
%keywords: buffer_type = if, then, else;
%sets
  letter = 'a'..'z' + 'A'..'Z';
  let_digit = letter + '0'..'9';
%fore
  $$
  buf:= '';
  $$
%productions
                  .
                  . {other productions}
                  .
[ident]: @intobuf, letter, (@intobuf, let_digit)*,
     @search;
                  .
                  . {other productions}
                  .
%actions
                  .
                  . {other actions}
                  .
  intobuf
  $$
  buf:= buf + upcase(chr(token_));
  $$

  search
  $$
  return_:= search_key(buf, ident);
  $$
                  .
                  . {other actions}
                  .
%end
```

Figure 9. Fragment of Lily code showing use of key words

values appropriately.

In the global part a type is declared which is later used in two other parts. First this type is used to declare the synthesized attribute of the example parser. Because of this declaration a buffer containing the actual string value for a key word or identifer will be returned by the parser as well as made available to the Pascal code blocks for synthesis. Second, the type defined in the global part is used to define the type of all key words in the key words part.

A non-default goal, "(ident)," is declared in the goals part of the parser. Corresponding to this an identifier structure is defined in the productions part defining an identifier as a letter followed by zero or more letters or numbers. In the course of parsing this structure an action called "@intobuf" is executed which constructs the buffer. Prior to each parse an action in the fore part initializes the value of buffer to the empty string.

The actual assignment of key word token values is done in the action at the end of the "(ident)" production. Here an action called "@search" is used to assign a return token value for the production. (Recall that non-default goals do not automatically assign such a value). The definition of the action in the actions part specifies that the standard variable "return_" is to be assigned the value returned from the Lily-generated search_key function. The buffer in parameter one of the function call is simply the buffer which the user has built to be compared against the key word list. In the second parameter the name of the non-default goal has been used as the value that the search_key function should return if the buffer fails to match a key word. (Recall that Lily automatically makes such constants available for non-default goals). This signifies that if the buffer does not contain a key word then the token is a simple identifier.

It should be noted that in creating the buffer, the standard Turbo Pascal "upcase" function has been used. This means that the buffer will always consist of uppercase letters and, consequently, the parser is case-insensitive. Had the "upcase" function been excluded the key words of the user-defined language would all have to be in uppercase since the search_key function only recognizes uppercase strings.

G. SETS PART

The sets part of a Lily parser specification is used to associate identifiers with sets of tokens. The part consists of the key word %SETS followed by one or more set definitions. A set definition consists of an identifier, an equals sign, and a set expression defining the contents of the set, terminated by a semicolon.

Lily set expressions are similar to those of Turbo Pascal although there are some extensions and simplifications. A set is composed of one or more set elements acted upon by zero or more operators. Set elements, exemplified in table VIII, can be goals or key words from the received parser, ordinals or ordinal ranges, or sets previously defined in the current parser. Expressions are evaluated from left to right according to operator precedence which may be overridden using parentheses. Set operators, together with their precedences and respective meanings are shown in table IX.

TABLE VIII

EXAMPLES OF SET ELEMENTS

Element		Meaning
'a'		ordinal = ASCII "a"
55		ordinal = 55
a_keyword	(identifier)	ordinal of key word (from received parser)
a_goal	(identifier)	ordinal of goal (from received parser)
a_set	(identifier)	contents of token set (from current parser)
'a'..'z'		Range of ASCII ordinals from "a" to "z"
55..75		Range of ordinals from 55 to 75

TABLE IX

EXPLANATION OF SET OPERATORS

Operator	Precedence	Meaning
- (unary)	1	Absolute complement *
*	2	Intersection
+	3	Union
-	3	Relative complement

* This is equivalent to relative complement with the universal set, U. Universal sets vary according to the cardinality of the token set in the received parser. For non-Lily and standard received parsers U = {x | x is an integer element of [0, 255]}. For Lily user-defined received parsers U = {x | x is an integer element of [0, c], where c = (number of goals of received parser + number of key words of received parser) - 1}.

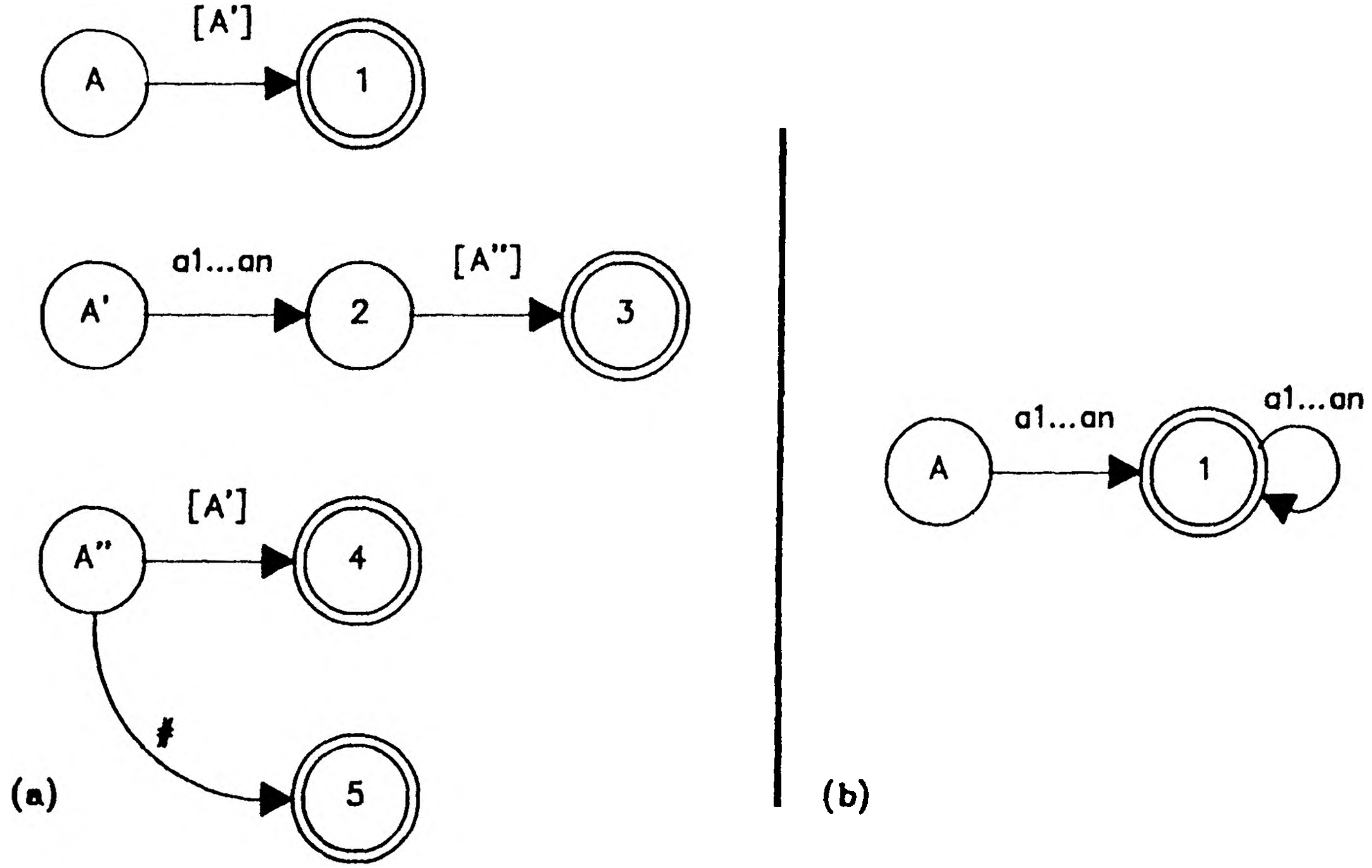

Figure 5. (a) Recursive RTN for structure inside "(" and ")+"
(b) Iterative RTN for same structure

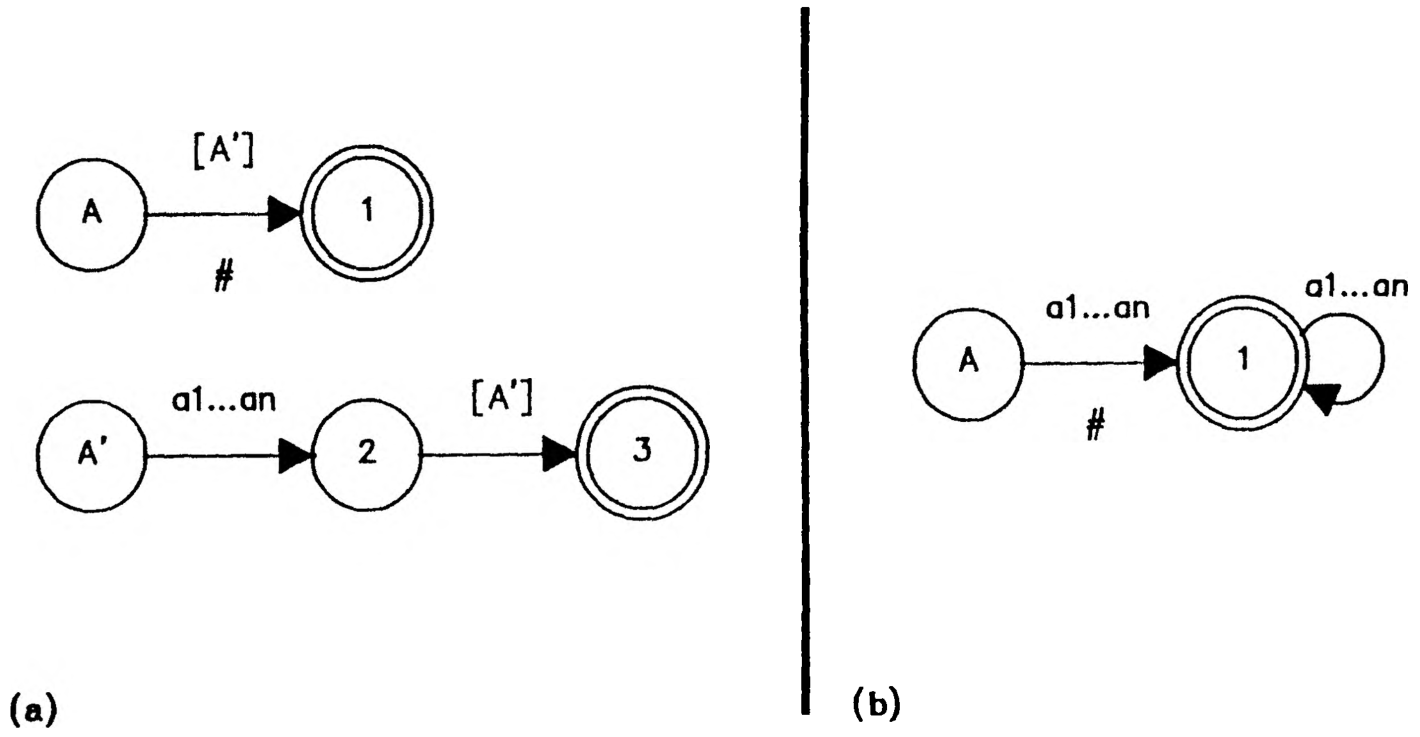

Figure 6. (a) Recursive RTN for structure inside "(" and ")*"
(b) Iterative RTN for same structure

Below are examples of the sets part:

```
%sets
letter = 'a'..'z' + 'A'..'Z';
consonant = letter - ('a' + 'e' + 'i' + 'o' + 'u' + 'A'
     + 'E' + 'I' + 'O' + 'U');
digit  = '0'..'9';
let_or_dig_or_under = letter + digit + '_';
uppercase = letter - 'a'..'z';
first_50 = 0..49;
twenty_to_60 = 20..60;
twenty_to_49 = first_50 * twenty_to_60;

%sets
relop = less + greater + equal + less_or_greater +
          less_or_equal + greater_or_equal;
         {assuming the above identifiers are defined
          as goals or keywords in the received parser}
```

Once a set has been defined it is available for use in subsequent set expressions as well as in production definitions. In such a definition the set name is considered a terminal symbol representing all terminal tokens contained in the set. Considering the set "digit" to be defined as above, the following two productions are equivalent:

```
[number]: (digit)+;

[number]: ('0' | '1' | '2' | '3' | '4' | '5' | '6' |
           '7' | '8' | '9')+;
```

Each production defines a string of digits such as might be used in the definition of an integer number.

Although the two productions given above are logically equivalent, nevertheless the two will result in different performances in the generated parsers. In general, a parser specification using sets will be more convenient to write and its generated parsing function will be somewhat smaller (roughly sixteen times the cardinality of the set in bytes is saved). On the other hand, a parser that uses the method in the second sample above will have a somewhat better time performance. The choice between the two alternatives is left to the discretion of the user.

H. FORE PART

The fore part is used to designate actions for the current parser to perform prior to beginning the parse. It is here that synthesized attributes and variables may be initialized.

The part consists of the key word %FORE followed by a Pascal code block containing a list of statements to be executed at the outset of the parse. Below is an example of a fore part:

```
%fore
$$
buffer:= '';
token_count:= 0;
$$
```

I. PRODUCTIONS PART

The productions part begins with the key word %PRODUCTIONS followed by a list of one or more production definitions. Each production definition is composed of: a nonterminal symbol identifier called the left side of the production; a colon separator; an expression called the right side of the production; and a terminating semicolon.

Production definitions, which may occur in any order, use an extended version of the Backus-Naur Form (BNF) to define the structure of the language recognized by the current parser. In each definition the nonterminal on the left side of the production is defined to have the structure represented by the expression in the right side. This expression consists of one or more symbols together with zero or more operators and parenthetic enclosures. Table X shows a table of grammatic symbols allowed in the right side of a production definition while table XI denotes the precedence of the two grammatic operators.

Three different forms of parenthetic enclosure are supported, each carrying a different meaning. Structures enclosed within "(" and ")*" are to be repeated <u>zero</u> or more times. This represents the Kleene closure of the structure. Structures enclosed within "(" and ")+" are to be repeated <u>one</u> or more times and this represents the transitive closure of the structure. Ordinary parentheses, "(" and ")" surround alternative lists from which exactly one alternative is to be selected. All of the parenthetic enclosures override the precedence of grammatic operators.

Identifier symbols in the right sides of productions consist of nonterminals (enclosed in brackets), terminal symbols, and action symbols (prefixed with "@"). The

TABLE X

EXAMPLES OF OPERANDS IN LILY PRODUCTIONS

Symbol		Meaning
'a'		character = ASCII "a"
55		ordinal = 55
a_keyword	(identifier)	key word terminal symbol (from received parser)
a_goal	(identifier)	goal terminal symbol (from received parser)
a_set	(identifier)	set terminal symbol (from current parser)
[nterm]	([identifier])	nonterminal symbol (from current parser)
@act	(identifier)	action symbol defined in current parser
#		null string constant

TABLE XI

EXPLANATION OF OPERATORS IN LILY PRODUCTIONS

Operator	Precedence	Meaning
, (comma)	1	Concatenation
| (sheffer)	2	Alternation

terminal symbols may be goals or keywords from the received parser or sets from the current parser. All nonterminals must be defined in the current productions part.

Characters and integer ordinals used in a right side are terminal symbols. Characters are intended to be used when the received parser (e.g. a Lily standard parser) supplies the current parser with tokens which are ASCII ordinal values. Integer ordinals should be used when an ASCII character value cannot be entered from the user's editor or when receiving non-Lily parsers.

The null string constant ("#") is used to signify the empty string. It may occur anywhere in the right side of a production. Action symbols are equivalent to the null string constant except that when they are encountered in the course of parsing a structure they perform some user specified-action. If this action requires the use of the current token then the action symbol should be written immediately <u>before</u> the terminal symbol of the token. All action symbols occurring in a productions part must be defined in the actions part of the parser specification.

Below are examples of the productions part.

```
%productions
[ident]:       letter, (letter | digit)*;
[period]:      '.';
[end_of_file]: 26;
[end_of_line]: 13, 10;

%productions
[expression]: [term], (addop, [term])*;
[term]: [factor], (mulop, [factor])*;
[factor]: left_paren, [expression], right_paren
        | minus_sign, [factor]
        | ident
        | digit_sequence;
```

The first example shows a productions part which might be used in the definition of a lexical analyzer. In this case each of the left side nonterminals would be declared as goals in the goals part of the parser.

The second productions part is typical of a parser for the higher-level syntax of a language. Here, an expression is defined in terms of tokens from the received parser, in this case a previously defined lexical analyzer. In addition, other nonterminals are used in the definition to break the language down into intelligible parts. At least one (perhaps the first) production is designated as a goal in the current parser's goals part.

J. NULL PART

The null part is optional and is used to define a structure which the current parser is to ignore. It is useful for defining the structure of comments as well as for designating the insignificance of other structures such as line-ending symbols or groups of blanks.

A null part is similar in structure to a production except that the left side is the key word %NULL. Following the key word is a colon, a grammatic expression, and a terminating semicolon. The following is an example of a null part:

```
%null: '{', (comment_char)*, '}'
     | (' ')+
     | 13, 10;
```

In this example three types of structures are defined as being insignificant and are to be discarded by the current parser. The first alternative defines anything enclosed within braces to be a comment. In the second alternative groups of blanks are to be ignored. And, finally, the third alternative specifies that the two-byte sequence of a carriage return and a line feed (used to denote the end of a line in ASCII text files) are insignificant in the current parser.

K. AFT PART

The aft part is optional and is used to encode actions to be taken just before exiting from the function generated for the current parser. This part can be used for, among other things, diagnosing errors.

The part consists of the key word %AFT followed by a Pascal code block. This block should contain a list of Pascal statements to be executed just prior to returning from the current parser's generated function.

Below is an example of the aft part.

```
%aft
$$
if return_ = error_ then writeln('Error.');
$$
```

L. ACTIONS PART

The actions part is optional but must be used if any action symbols are present in the productions of the current parser. This part allows the user to associate Pascal code blocks with action symbols. Each time the parser enters a state characterized by an action symbol the corresponding Pascal code block will be executed.

An actions part consists of the key word %ACTIONS followed by a list of action definitions. Each definition consists of an identifier, associated with an action symbol in the productions part, followed by a Pascal code block defining the action to be taken. All action symbols used in the productions part must be resolved in this way or Lily will flag an error.

Suppose that two action symbols "@intobuf" and "@check" have been used in the productions part of the current parser. These two actions might be resolved in the actions part according to the example below.

```
%actions
intobuf
$$
buffer:= buffer + chr(token);
$$

check
$$
if length(buffer) > max_length then
  begin
  writeln('Buffer ', buffer,' exceeds max length.');
  return_:= error_;
  state_:= accept;
  end;
$$
```

M. DISAMBIG PART

The disambig part is used to override the operation of the parser. It can be used for resolving ambiguities in the user's language structure as well as for adding error recovery and diagnostic capabilities to the current parser.

A disambig part consists of the key word %DISAMBIG followed by a list of overriding state-token transitions, each with an associated Pascal code block. The state token transitions consist of an integer constant for a state, then an integer constant or null string constant ("#") representing, respectively, a token number or "any token," and, finally, an integer constant representing the next state. Each of these three fields in a state-token transition are separated by commas. Following each state-token transition is a Pascal code block defining an action to be carried out whenever the state-token pair is encountered in the course of a parse.

Transitions specified in the disambig part have a higher priority than transitions automatically generated by Lily. This means that as Lily constructs a parsing function for the current parser, state-token transitions entered in the disambig part will override the corresponding transitions generated by Lily.

It is important to note the relationship between the productions part and the disambig part. Lily numbers production symbols beginning with the production of the null part and ending with the last symbol of the productions part. The disambig part uses these numbers in specifying state-token transitions. Because of this any change to the productions part prior to a production which has been disambiguated will result in the necessity of changing the disambig part. While this is sometimes inconvenient, a good rule of thumb can minimize the necessity for making such changes. Whenever a production is known to contain ambiguities that production should be moved to the top of the production list.

Examples of a syntactically correct disambig parts follow.

```
%disambig
  0, 58, 14
  $$
  if look_getchar_l(1) = 61 then state_:= 11;
  action_:= shift_;
  $$

%disambig
  0, 0, 3
  $$
  if look_nothlex(1, buffer, lngth, subtoken) = 6 then
    begin
    state_:= 5;
    push_(4);
    end
  else push_(2);
  $$
```

In the first example, whenever the parser is in state 0 and looking at token 58, it will assume a transition to state 14. Then an action is executed which uses look-ahead. If the next token to be seen is a 61, then the state of the parser is changed from 14 to 11. Finally, the parser is told to execute a shift.

The second example is similar except that after using look-ahead to decide which state is next, separate "push_" actions are done. In practice, the part of the action exemplified by the calls to the push_ procedure is copied from the action procedures of the generated parser. This process is explained in more detail in the next chapter, section B, subsection 3.

II. USING THE LILY SYSTEM

The current chapter contains information about actually running the Lily metacompiler and using the two supplemental minimal perfect hash table generators by Dr. Thomas J. Sager. In addition, this chapter provides information on the creation, disposition, and interpretation of the various files produced by the two systems.

A. RUNNING THE SYSTEM PROGRAMS

The current version of Lily is set up to run on the IBM PC under the Personal Computer Disk Operating System (PC-DOS). On this architecture Lily will normally require 256 kilobytes of memory and at least one disk drive. Some augmentation to the DOS system configuration may be necessary on some runs as Lily works with a considerable number of files and may require more than the default number of buffers (two) or files (eight). Increasing the available buffers or file handles can be accomplished by modifying the CONFIG.SYS file as described in the DOS manual. Both the BUFFERS and FILES specifications should be changed to fine tune the system to its optimal speed.

The following files make up the Lily system:

1.	LILY.COM	(the parser generator)
2.	LILY.000	(an overlay for LILY.COM)
3.	PARHASH.COM	(hash program for tables)
4.	KEYHASH.COM	(hash program for key words)
5.	SEARCH.PAS	(code for hashing key words)

Once a Lily system user has written a file of Lily source code the next step will be to process the file using the parser generator, LILY.COM. Next, any parser table files generated by the metacompiler will be processed by the table-to-hashing function program PARHASH.COM. Finally, any key word files will be processed by KEYHASH.COM, the key word hashing function generator.

The above process is best illustrated using an example. For the subsections that follow consider the source code of figure 15 (back of appendix) to be in a DOS file called NOTHING.LIL. The source file is a file of text containing no special characters such as tab characters, etc..

1. The Lily Program: In the first step of processing a Lily program the user will run LILY.COM, the parser generator. This can be done by simply typing LILY at the DOS prompt. The program will be loaded and the user will be prompted for the following items:

1. The source file name: The user enters NOTHING.LIL or simply NOTHING.
2. The target file name: A name is entered for the destination of the Pascal code generated by LILY. If a simple carriage return is entered, the default name is NOTHING.PAS.
3. The information file name: A name is entered for the destination of the Lily information file. A carriage return accepts the default NOTHING.INF.
4. The listing file name: A name is entered for the destination of the listing file. A carriage return accepts the default NOTHING.LST.
5. The drive destination for tables: A letter is entered for the drive to which .TBL, .RST, .KWD, and .PRC files will be sent. These files will be placed on the current directory of the selected

drive. A carriage return accepts the logged drive as a default.

If the system user wishes to accept all of the default destinations, then a simpler procedure for the above may be used. Lily accepts a single parameter at the command line. To accept all of the default destinations the user may type either LILY NOTHING.LIL or simply LILY NOTHING.

Lily will proceed to process the source code in NOTHING.LIL and will report its progress on the screen. If at any point the user wishes to interrupt the processing, he may do so by striking any key.

After Lily finishes operating, a number of new files will have been created. The names of the created files along with their associated meanings may be found in the Lily information file, a file of text given the name NOTHING.INF in our example run. Assuming that the user chose to accept the default names the new files will be as follows:

1. NOTHING.PAS
2. NOTHING.INF
3. NOTHING.LST
4. NOTHING.RST
5. NOTHLEX.TBL
6. NOTHLEX.PRC
7. NOTHLEX.KWD
8. NOTHPARS.TBL
9. NOTHPARS.PRC

The .TBL, .PRC, and .KWD files are named by appending the appropriate extension to the first eight characters of the names given in the parser headings of the source. Thus, NOTHLEX.TBL contains the parsing table for the parser "nothlex" in the source code. NOTHLEX.PRC contains the action procedures for the same parser while NOTHLEX.KWD contains that parser's key words. The parser "nothpars" in the source has, in like manner, a .TBL file and a .PRC file but, since the parser has no key words, no .KWD file.

The file NOTHING.RST contains a reset procedure written in Pascal. This procedure is used to return the generated parsing functions to their initial state. This file can be included by the user in his driver program or discarded.

The file NOTHING.PAS contains the Pascal code implementing the parsers in the source file. It contains code for the functions "nothlex" and "nothpars" in addition to code for the standard low-level parser "getchar_1." Also included are the two look-ahead functions requested in the source code. These two functions will be called, respectively, "look_getchar_1" and "look_nothlex". The two .PRC files discussed above are parts of the code of NOTHING.PAS through Turbo Pascal file inclusion (using the $I directive) as are two files yet to be generated: the hashing function tables NOTHLEX.HSH and NOTHPARS.HSH.

The two .HSH files are generated using the minimal perfect hash function generation programs PARHASH.COM and KEYHASH.COM. To accomplish this the user will first process the .TBL files. This is done using PARHASH.COM.

2. The Parhash Program: The user executes the program by typing PARHASH at the DOS command line. When prompted for a

source file the user responds with NOTHLEX.TBL. The program will then ask if the user wishes progress information sent to the screen or the printer. After a response is entered the program will proceed to calculate the hashing function for the parsing table in NOTHLEX.TBL. Finally, if everything goes well, the user is prompted for the name of the output file. Entering a carriage return accepts the default of NOTHLEX.HSH.

Upon completing the above operation the user will repeat the process for NOTHPARS.TBL and, in general use, for all Lily-generated .TBL files. As in the LILY program, the PARHASH program can accept a source file name as a command line parameter. This means that a more convenient method for accomplishing the above is to simply type PARHASH NOTHLEX.TBL. If this is done, then screen (as opposed to printed) progress reports and output file NOTHLEX.HSH are chosen by default.

3. The Keyhash Program: The final phase of processing involves the use of the KEYHASH program to generate a minimal perfect hash table for keywords. This is a process similar to using the PARHASH program but entailing additional considerations about the structure of the key words. The reason for this difference is explained below.

The user enters the name KEYHASH at the command line and is prompted for the following:

1. The source file name: Here the user enters NOTHLEX.KWD.

2. Screen as opposed to printed progress reports: The user enters an 'S'.

3. Whether to accept the default number of vertices: The user enters a 'Y' to accept the default. An 'N' causes the program to prompt the user for a number to use. This is explained below.

4. The name of the target file (at end of processing): The user enters a name or a carriage return to accept the default name NOTHLEX.HSH.

5. Whether to append to an existing file, choose a new name or overwrite the existing file: The user chooses "append" since part of the file NOTHLEX.HSH was created previously by the PARHASH program. The new hashing table is properly appended to the old.

As with the PARHASH program the KEYHASH program accepts a single parameter at the command line. This means that for our example, the user could run the KEYHASH program by simply typing KEYHASH NOTHLEX.KWD. Unlike the PARHASH program, however, the KEYHASH program still prompts the user for additional information, even if the command line parameter is used.

The processing of key words involves elements in both the LILY program and the KEYHASH program. A file, called SEARCH.PAS, on the Lily system disk contains a fragment of Pascal code which defines a pseudo-random function over the letters of key words. This function is intended to provided a series of three values for each key word which must be distinct from the corresponding series in any other key word. If the function computes the same series for two different key words, then a hashing collision occurs and the KEYHASH program will fail to create a hash table. For this reason it may be necessary to alter the function.

The function defined in the SEARCH.PAS file supplied with the Lily system (see figure 10) will generate a series of three values for each key word. This will be unique provided that no two key words have the same first, the same last, and the same middle letter (i.e. the letter in position number = length(word) div 2). If these conditions are met then the KEYHASH program should function correctly and produce the required hash table once a suitable number of vertices has been determined.

The number of vertices used by the KEYHASH program is given a default value which the user may alter. Such alteration can be done for two reasons: to reduce the number of vertices used in order to conserve space, or to provide a substitute for a default value which fails to work successfully in the operation of the program.

When the KEYHASH function fails to generate a hashing table the user should first examine his key words. If the function fragment in SEARCH.PAS does not provide a unique series of three numbers for each key word, then it will be necessary to modify the function fragment. Once this is done it will be necessary to recompile the KEYHASH program and start at the beginning of the metacompilation process by running the LILY program again.

```
var len: integer;
begin
len:= length(inword);
h0val:= ord(inword[len shr 1]) mod vert_mod_;
h1val:= ord(inword[1]) mod vert_mod_;
h2val:= ord(inword[len]) mod vert_mod_ + vert_mod_;
```

Figure 10. Hashing fragment in SEARCH.PAS file*

*This fragment forms part of the Lily system. Should it be necessary to modify this fragment the new fragment must be similar in structure to the old. I.e. variables may be declared, followed by a "begin", and ending with code for making assignments to h0val, h1val, and h2val. Assignments to h0val and h1val should made using the standard modulus vert_mod_ as above. Assignments to h2val should be done, correspondingly, using vert_mod_ as a modulus and then adding vert_mod_. No "end" occurs at the end.

If the function fragment in SEARCH.PAS succeeds in generating three unique values for each key word but the KEYHASH program still fails on the first attempt, the system user should simply rerun the KEYHASH program, using different numbers of vertices until success is achieved.

B. WRITING LILY PROGRAMS

The best way to learn to write Lily programs is to begin by studying examples of other programs. Toward that end figures 15, 16, and 17 at the end of this chapter contain tested examples of Lily source code. In addition to studying the examples, the system user will find it helpful to turn to the syntactic and semantic information given in the previous chapter.

Lily is a program for generating parsing tables and a series of parsing functions driven by these tables. These functions, written in Turbo Pascal, all return type integer and take parameters as specified in the parser attributes of the function's Lily parser specification. A return value of -1 from any of these functions connotes error while any non-negative return value indicates that a corresponding structure in the source was recognized.

1. Compiler Implementation Strategies: In general, a system user wishing to write a Lily program should begin by considering the structure and size of the language to be implemented. Lily is designed to assist in implementing

LL(1) language compilers and those structures in the user's language which are not LL(1) should be identified. Additionally, the size of the user's language will play a part in deciding how the compiler front end is to be partitioned. The standard method of breaking a compiler into lexical and syntactic analysis phases is highly recommended and is in fact the only method which has been tested using Lily.

Other methods which may be used to implement larger front ends include the suggestions of extending the front end of the compiler into more than two distinct phases or using one lexical analyzer feeding tokens into a group of parsers. The first method can be called "segmenting the compiler vertically" while the second method can be called "segmenting the compiler horizontally."

In vertical segmentation the lexical analyzer will feed tokens to a higher-level analyzer. Then this analyzer will, in turn, feed tokens into a still higher-level parser and so on. This method forms a chain of parsers rising in complexity from a low-level analyzer to a single, final high-level parser. For example, a lexical analyzer could break source code into conventional tokens while a mid-level parser could group these tokens into structures of medium complexity, such as expressions, lists of identifiers, or even whole programs. A final parser can then group the mid-level structures into a single final structure constituting a program or group of programs.

In contrast, horizontal segmentation, is simply a way of dividing the parsing chores among a group of parsers. A single lexical analyzer provides tokens for the group. Then a driver program chooses a parser to use on the received tokens based on the first token in the list or on the current context. In this way an entire parser could be dedicated to each part of a language. For example, one parser might be applied to declaration structures while another is applied to statement lists.

Usually, however, the accepted division of a compiler front end into two phases will be the method of choice. The other two methods are only mentioned to allow the Lily system user to implement front ends for larger languages. Under these circumstances Lily might not be able to handle the large parsers implied by a two-phase structure.

Writing a Lily program should be a step-wise process. Each parser specification should be developed and tested before moving on to the next. In practice this will mean writing and testing, first, the lexical analyzer, followed by the parser. This allows the Lily system user to be sure of one phase before attempting to move on to another.

For instance, after writing and successfully metacompiling a Lily specification for a lexical analyzer, the user should test the generated function. This can be done by writing a short driver program around the generated function and supplying a file of input. Often this will allow users to spot errors in their productions.

Once a parser has been metacompiled and tested, the creation of hash files for that parser will no longer be necessary. Those hash files previously generated can be used without fear. This means that the programs for creating minimal perfect hash functions, described earlier, need only be used to create the tables for untested parsers.

2. Error Messages From Lily: In the course of running LILY.COM, the Lily parser generator, a number of errors in the user's source may be detected. Some of these are fatal and will terminate the operation of the parser generator. Warning-level messages, on the other hand, merely flag irregularities as they occur. Those irregularities that are sufficiently significant may result in a request for user intervention.

The fatal error messages are shown in table XII. These errors are primarily concerned with program syntax and semantics. In addition to the error message given, diagnostic and trouble-shooting messages will sometimes be included detailing where the error occurred and how the it may be corrected.

Warning-level messages are shown in table XIII. For those warnings requesting user intervention the extent of intervention possible is either the termination or the continuation of the parser generator operation. Under most circumstances it is best to allow the program to continue so that the Lily information file may be completed. This

TABLE XII

FATAL ERRORS IN A LILY PROGRAM

Error Message	Meaning
Lexical error.	A token of the language is incorrect or contains an illegal character.
(token) expected, (place)	The token on the left was expected in the language part on the right.
Goal (name) is undefined ... parser (name).	A goal in the goal list is not defined in the production list of the current parser.
Action (name) is unresolved ... parser (name).	An action is used in the productions part but not defined in the actions part.
Action (name) is multiply defined ...	An action in the actions part is defined more than once.
Terminal (name) is undefined ...	A terminal identifer used in a production or set is not a goal or keyword of the received parser or a set of the current parser.
Semantic error in integer ordinal ...	An integer ordinal was outside the allowed range.
Nterminal (name) is multiply defined.	A nonterminal occurs in more than one left side.
Identifier (name) is previously declared ...	An identifier has been previously declared. Occurs in goal, key and set names.
Parser (name) collides in first 8 characters with parser (name)	Two parser names were identical in the first eight characters.

TABLE XIII

WARNINGS

Warning	Meaning
Nonterminal (name) useless in parser (name). It has been discarded.	A nonterminal is not on a path from some goal. This means it will never be used.
Grammar is left-recursive in nonterminal (number).	A nonterminal is left recursive. I.e. it can produce itself without an intervening prefix.
Grammar of parser (name) has too many automata. ...	The grammar has more than 490 symbols.
Production (name) ... cannot generate terminal strings.	A production is useless.
Action (name) is defined but not used.	An action in the actions part was never used in the productions part
Parser (name) in receives part of parser (name) is not defined in a Lily parser-spec.	Warning issued when the user specifies a non-Lily parser in the current parser's receives part.
Ambiguity. State (number) Token (number or #) Next (number) Action (number) kept. Same state-token pair with Next (number) Action (number) discarded.	A state contains an ambiguous transition. The grammar will need to be altered or the ambiguity will need to be resolved in the disambig part.
Set-token ambiguity. ...	A state contains a transition on a set and a token but the set contains the token.
Set-set ambiguity. ...	A state contains a transition on two sets but the sets are not disjoint.
Alternative beginning with state (number) is nullable inside a repetetive closure ...	Such alternatives can result in endless looping.

file can then be used to help correct the error.

3. Resolving Ambiguities: Ambiguities result when a single token is a prefix of more than one alternative among a list of alternatives or of more than one goal in a list of goals. The result of ambiguities in a parser is the existence of states which, on a single token, require the parser to perform more than one distinct transition. Lily detects ambiguities and issues appropriate warnings to the user, both on the screen and in the Lily information file.

Ambiguities in Lily fall into three classes: token-token ambiguities (referred to simply as ambiguities), set-set ambiguities, and set-token ambiguities. Each class is resolved differently, in some cases automatically.

The Lily information file generated for each Lily source program contains data which may be of assistance in resolving ambiguities. A fragment from a Lily information file is shown in figure 11. Table XIV contains a legend which explains the coding method used in the fragment. Automaton numbers shown in the table may be considered "before" states in the action of the parser; the parser is in these states just prior to seeing the structure in the right side. Each state number coded in a right side may be considered an "after" state; the parser will be in one of these states just after seeing the corresponding symbol.

Set-set ambiguities occur when two non-disjoint sets conflict in an alternative list. This situation is shown

```
Nonterminal:   %NULL
is a goal returning 255 mode = Non-default
Automaton Number:    1
...is defined as
 '{' {2,123},( {7} COM_CHAR{s} {8,259})*,'}' {9,125}|
( {3} ' ' {4,32})+|
^13 {5,13},^10 {6,10}


Nonterminal:   CONSTPART
is a non-goal.
Automaton Number:    24
...is defined as
 CONST{k} {33,22},( {35} [CONSTDECL] {36,37})+|# {34}

Nonterminal:   CONSTDECL
is a non-goal.
Automaton Number:    37
...is defined as
 IDENT{g} {38,0},@CHECKEQUAL {39},RELOP{g} {40,16},
 [CONSTANT] {41,42}, SEMICOLON{g} {43,13}
```

Figure 11. Example of productions in the Lily information file

TABLE XIV

LEGEND FOR PRODUCTIONS IN THE LILY INFORMATION FILE

Symbol	Meaning	Code suffix
identifier{g}	goal	{state, token}
identifier{k}	key word	{state, token}
identifier{s}	set	{state, token}
[identifier]	nonterminal	{after state, before state}
@identifier	action	{state}
^number	integer ordinal	{state}
'c' (c arbitrary)	character ordinal	{state}
#	null string	{state}
(	enclosure	{state}

in figure 12a. When such set-set ambiguities occur Lily will automatically assume that an incoming token, logically belonging to both sets, belongs to the first set in the parser's set list. In our example this means that a "9" will be considered a "digit" before it will be considered a "letter_or_digit." If such a default assumption fails to produce a correct parser, then the user can still resolve the problem by excluding the conflicting elements from one of the sets.

Set-token ambiguities occur when a set and a token, contained by the set, conflict in an alternative list. Figure 12b shows an example of a set-token ambiguity. When this circumstance arises Lily gives priority to the token. In our example this means that an "a" will be considered an "a" before it will be considered a "letter." If this method of resolution fails to produce a correct parser then the problem can be resolved by excluding the token from the set.

Other ambiguities, caused by conflicts between tokens in an alternative list, may be resolved using the disambig part in the Lily program. To accomplish this the following three-step procedure is recommended:

1. Examine the Lily information file to determine which productions contain ambiguities. Move these productions to the top of their productions part and run LILY.COM again to learn the new ambiguous state numbers.

2. Add ambiguity-resolving actions to the parser using some criteria for choosing between ambiguous transitions. Look-ahead is one possible criterion.

3. Process the new specification and test the

(a)

```
.
.
.
%sets
digit = '0'..'9';
letter_or_digit = 'a'..'z' + 'A'..'Z' + digit;
.
.
.
%productions
[set_set_ambiguous]: digit | letter_or_digit | ... ;
.
.
.
```

(b)

```
.
.
.
%sets
letter = 'a'..'z' + 'A'..'Z';
.
.
.
%productions
[set_token_ambiguous]: 'a', 'b', 'c' | letter | ... ;
.
.
.
```

Figure 12. (a) Example of set-set ambiguity
(b) Example of set-token ambiguity

generated parser function to ensure that it functions properly.

Step one involves observing the ambiguity warnings for a parser. These are given at the top of the Lily information file. Then, by matching the state numbers in the warnings to coded productions in the information file, the productions containing the ambiguities may be determined. Once this is accomplished, those productions should be moved to the top of their productions part in the Lily specification. This is done for reasons explained in the previous chapter under section L on the disambig part.

The second step is best explained by example. The Lily specification for the language Nothing (figure 15 at the end of this chapter) contains ambiguities of two common types: a conflict between alternate terminal symbols, and a conflict between alternate nonterminal symbols. The process for resolving each of these two types is slightly different.

Deleting the disambig parts of the Nothing specification and running LILY.COM will result in ambiguity warnings at the screen and in the Lily information file. (Note that only the first transition disambiguation was deleted from the disambig part of Nothlex.) These warnings are shown in figure 13. Figure 14 shows the productions part information corresponding to the states indicated in these warnings. Note that these productions have already been moved to the top of their respective productions parts. The disambig parts in the parser specifications for Nothing correct these

```
.
.
.
Warning.  Ambiguity in parser»NOTHLEX
State»0 Token»58 Next»11 Action»-2 kept.
Same state-token pair with Next»14 Action»-2 discarded.
.
.
.
Warning.  Ambiguity in parser»NOTHPARS
State»0 Token»0 Next»3 Action»1 kept.
Same state-token pair with Next»5 Action»2 discarded.
.
.
.
```

Figure 14. Warnings after making deletions to the disambig parts in NOTHING.LIL (figure 15) *

* Specifically, the first half of the disambig part of Nothlex and the entirety of the disambig part of Nothpars were deleted.

```
Productions as follows: 1

Nonterminal:   ASSIGN
is a goal returning 10 mode = Default
Automaton Number:    10
...is defined as
 ':' {11,58},'=' {12,61}

Nonterminal:   COLON
is a goal returning 14 mode = Default
Automaton Number:    13
...is defined as
 ':' {14,58}

Productions as follows: 2

Nonterminal:   FACTOR
is a non-goal.
Automaton Number:    0
...is defined as
 LEFT_PAREN{g} {1,6},[EXPR] {13,14},RIGHT_PAREN{g}{15,7}
| [VARIABLE] {2,3}
| [FUNCTION_CALL] {4,5}|INTGR{g} {6,1}
| CHARACTER{g} {7,2}|CRET{k} {8,43}
| FALSE{k} {9,41}|TRUE{k} {10,42}
| NOT{k} {11,37},[FACTOR] {12,0}

Nonterminal:   VARIABLE
is a non-goal.
Automaton Number:    3
...is defined as
 IDENT{g} {145,0},[INDEXLIST] {146,147}

Nonterminal:   FUNCTION_CALL
is a non-goal.
Automaton Number:    5
...is defined as
 IDENT{g} {211,0},LEFT_PAREN{g} {212,6},[ARGLIST] {213,214},
RIGHT_PAREN{g} {215,7}

Nonterminal:   INDEXLIST
is a non-goal.
Automaton Number:    147
...is defined as
 LEFT_BRACKET{g} {216,8}, ..., RIGHT_BRACKET{g} {219,9}|#
{217}
```

Figure 14. Production information for states in warnings of figure 13

1 Taken from productions information of Nothlex
2 Taken from productions information of Nothpars

ambiguities.

In the specification for Nothlex, the lexical analyzer, two goals share a common prefix, a colon. The disambig part for Nothlex shows the use of look-ahead to resolve these ambiguities. If the colon is immediately followed by an equals sign, the parser is instructed to make a transition to the state for a Nothing assignment operator. Otherwise, the parser makes a transition to the state for a simple colon.

The actions to be taken following these transitions are user-defined by referring to the ambiguity warnings. Basically, these actions simply carry out the actions which would have taken place were there no ambiguities in the parser specification. In the case of Nothlex, the warnings show action numbers of -2 whichever transition is made. When action numbers are less than zero, these actions are specified in the disambig part by simply assigning the standard integer variable "action_" to the value given in the warning.

Ambiguity resolution in the Nothpars specification is only a little more complicated. In this case the parser needs to be told how to choose between transitions for two nonterminals. Specifically, an action is needed to decide when the parser is looking at a variable and when it is looking at a function call. The disambig part of Nothpars makes this decision using look-ahead. If an identifier is followed by a left parenthesis then the parser makes a transition corresponding to a function call. Otherwise, a transition corresponding to a variable is made.

Notice, however, that the action numbers shown in the ambiguity warning for Nothpars are both non-negative. When this occurs the user must ensure that these actions are carried out after the transition decision has been made. To accomplish this the user, somewhat inconveniently, looks up the action procedures corresponding to the action numbers. These procedures are found in the .PRC file (in this case, NOTHPARS.PRC), where they are given procedure names corre-sponding to their action numbers. The user simply copies the statement list found in the action procedure. General-ly, this will mean nothing more then writing a "Push_" or "Pop_" depending on which transition is made.

4. Driving Lily-Generated Parser Functions: The parsers generated by Lily are Pascal functions. In order to parse a string with these functions it is only necessary to set up a driver program to make the appropriate function calls. The Lily-generated functions are then incorporated into the driving program's source code.

Usually the lowest-level parsing function will receive its input from an ASCII text file as has been remarked earlier in this document. If this function, perhaps a lexical analyzer, makes use of the standard Getchar func-tions, the user will have to establish the necessary interface between the Getchar function and the source or listing files. In practice this should mean no more than assigning DOS names to the standard files needed by the Getchar function. Then the files should be opened: for reading in the case of the input file; for writing in the case of the listing file (if such is required). In addition, if the user has not declared the logical file names, INFILE and LISTFILE, in the global part of his Lily specification, these will have to be declared in the driver program.

For example, suppose that a user has defined a lexical analyzer called Lex which receives GETCHAR_L. The user must make sure that his driver program contains a file declaration for the two files which will be used by GETCHAR_L: INFILE and LISTFILE. Before calling the Lex function to begin lexical analysis, the user must also assign DOS names to the files and make sure that they are opened. INFILE is opened for reading; LISTFILE is opened for writing.

Finally, the calling of the Lily parsers themselves is done in a very natural way. They are simply Pascal functions returning integer type. The actual parameters supplied to these functions correspond in type, number, and order to the attributes listed in the parser headings of the Lily specifications.

```
%global
  $$
const charbufsize = 20;
var input_file, list_file: text;
type charbuf = string[charbufsize];
  $$
                 {request for standard parser}
%parser getchar_l %look 1;

                 {Lexical analyzer for Nothing}
%parser nothlex(<buffer: charbuf; <lngth,
    <subtoken: integer) %look 1;
  %receives getchar_l;
  %local
    $$
  var i: integer;
    $$
  %goals
  (ident), intgr, character, asterisk, slash, ampersand,
  left_paren, right_paren, left_bracket, right_bracket,
  assign,   ellipsis, comma, semicolon, colon, period,
  relop, plus, minus, sheffer, end_of_file;

  %keywords: charbuf =
  program, const, type, array, of, var, function, begin,
  end, if, then, else, while, do, read, write, not, integer,
  char, boolean, false, true, cret, return;

  %sets
    letter      = 'a'..'z' + 'A'..'Z';
    digit       = '0'..'9';
    let_digit   = letter + digit;
    com_char    = - (26 + '}');    {no eof (26) in comments}
    universe    = 0..255;

  %fore              {initializes attributes}
    $$
    buffer:= '';
    lngth:= 0;
    subtoken:= 0;
    $$

  %productions
    {Note that goals [assign] and [colon] share a colon as
     a common prefix.  Also goals [ellipsis] and [period]
     share a period as a common prefix.  These four are
     placed at the top to facilitate disambiguation.}
    [assign]:   ':', '=';
    [colon]:    ':';
```

Figure 15. Lily code for the language Nothing

```
    [ellipsis]: '.', '.';
    [period]:   '.';
    [ident]: @intobuf, letter, (@intobuf, let_digit)*,
      @search_key;
    [intgr]: (@intobuf, digit)+;
    [character]:     ''', @intobuf, universe, ''';
    [asterisk]:      '*';
    [slash]:         '/';
    [ampersand]:     '&';
    [left_paren]:    '(';
    [right_paren]:   ')';
    [left_bracket]:  '[';
    [right_bracket]: ']';
    [comma]:         ',';
    [semicolon]:     ';';
    [relop]:          '<', ('>', @ne | '=', @le | @lt) |
                      '>', ('=', @ge | @gt)
                      '=', @eq;
    [plus]:          '+';
    [minus]:         '-';
    [sheffer]:       '|';
    [end_of_file]:    26;

%null: '(', (com_char)*, ')' | (' ')+ | 13, 10;
%actions
  search_key
    $$
    return_:= search_key(buffer, ident);
    $$
  intobuf
    $$
    lngth:= lngth + 1;
    buffer:= buffer + upcase(chr(token_));
    $$
  eq $$ subtoken:= 0; $$
  ne $$ subtoken:= 1; $$
  lt $$ subtoken:= 2; $$
  gt $$ subtoken:= 3; $$
  le $$ subtoken:= 4; $$
  ge $$ subtoken:= 5; $$
%disambig
  0, 58, 14
  $$          (* disambiguate [assign], [colon] *)
  if look_getchar_1(1) = 61 then state_:= 11;
  action_:= shift_;
  $$
```

Figure 15. (continued) Lily code for the language Nothing

```
  0, 46, 19
  $$          (* disambiguate [ellipsis], [period] *)
  if look_getchar_1(1) = 46 then state_:= 16;
  action_:= shift_;
  $$


%parser nothpars;
%receives nothlex;
%goals
  program;
%productions
   (Note that the ambiguous definition for [factor] has
    been moved to the top of the productions part.  This
    facilitates disambiguation as above.)

 [factor]: left_paren, [expr], right_paren | [variable] |
   [function_call] | intgr | character | cret | false |
   true | not, [factor];
 [program]: program, ident, semicolon, [block], period;
 [block]: [constpart], [typart], [varpart], [funcpart],
   [execpart];
 [constpart]: const, ([constdecl])+ | #;
 [constdecl]: ident, @checkequal, relop, [constant],
   semicolon;
 [constant]: [sign], intgr | character;
 [sign]: plus | minus | #;
 [typart]: type, ([typdecl])+ | #;
 [typdecl]: ident, @checkequal, relop, array, left_bracket,
   [rangelist], right_bracket, of, [simptype], semicolon;
 [simptype]: integer | char | boolean;
 [rangelist]: [range], (comma, [range])*;
 [range]: [bound], ellipsis, [bound];
 [bound]: [sign], intgr | character;
 [varpart]: var, ([vardecl])+ | #;
 [vardecl]: [identlist], colon, [type], semicolon;
 [type]: [simptype] | ident;
 [identlist]: ident, (comma, ident)*;
 [funcpart]: ([funcdecl])*;
 [funcdecl]: function, ident, [parmpart], colon, [simptype],
   semicolon, [block], semicolon;
 [parmpart]: left_paren, [parmdecl], (semicolon,
   [parmdecl])*, right_paren;
 [parmdecl]: [identlist], colon, [type];
 [execpart]: begin, [stmtlist], end;
 [stmtlist]: ([stmt])*;
 [stmt]: [asstmt] | [ifstmt] | [whstmt] | [readstmt] |
   [writestmt] | [retstmt];
 [asstmt]: [variable], assign, [expr], semicolon;
 [variable]: ident, [indexlist];
 [ifstmt]: if, [expr], then, [stmtlist], [iftail];
```

Figure 15. (continued) Lily code for the language Nothing

```
 [iftail]: else, [stmtlist], end | end;
 [whstmt]: while, [expr], do, [stmtlist], end;
 [readstmt]: read, left_paren, [inlist], right_paren,
   semicolon;
 [inlist]: [variable], (comma, [variable])*;
 [writestmt]: write, left_paren, [outlist], right_paren,
   semicolon;
 [retstmt]: return, left_paren, [expr], right_paren,
   semicolon;
 [outlist]: [expr], (comma, [expr])*;
 [expr]: [simpexpr], [exprtail];
 [exprtail]: relop, [simpexpr] | #;
 [simpexpr]: [sign], [term], ((plus | minus | sheffer),
   [term])*;
 [term]: [factor], ((asterisk | slash | ampersand),
   [factor])*;
 [function_call]: ident, left_paren, [arglist], right_paren;
 [indexlist]: left_bracket, [outlist], right_bracket | #;
 [arglist]: [outlist];

%actions
  checkequal $$
  if subtoken <> 0 then
    begin
    writeln('Error. Equals expected.');
    return_:= error_;
    state_:= accept_;
    end;
  $$

%disambig
  0, 0, 3
  $$         (* disambiguation for [factor] *)
  if look_nothlex(1, buffer, lngth, subtoken) = 6 then
    begin
    state_:= 5;
    push_(4);
    end
  else push_(2);
  $$

%end
```

Figure 15. (continued) Lily code for the language Nothing

```
%global
  $$
  const
    max_string = 20;
  type
    charbuf = string[max_string];
  var list_file, input_file: text;
  $$
          {request a standard parser}
%parser getchar_1 %look 2;
          {lexical analyzer for Lily}
%parser lil2lex(<buffer: charbuf; <lngth: integer);

%receives getchar_1;

%local
  $$
  var i: integer;
  int_con_too_long: boolean;
  dummy: Integer;$$

%goals
  term_ident, nterm_ident, syn_att_ident, inh_att_ident,
  act_sym_ident, int_con, left_paren, rt_paren_ast,
  rt_paren_plus, rt_paren,  colon, semicolon, sheffer,
  comma, ellipsis, equals, character, pascal_stuff,
  asterisk, plus,   minus, null_string, end_of_file,
  (keyword);

%keywords: charbuf =
  parser, goals, actions, receives, sets, null, end,
  keywords, disambig, global, productions, local, fore, aft,
  look;

%sets
  letter        = 'a'..'z' + 'A'..'Z';
  digit         = '0'..'9';
  let_num_dash  = letter + digit + '_';
  comment_char  =  - ( 26 + '}');  {no eof (26) in comments}
  universe      = 0..255;

%fore         {initializes attributes}
  $$
  lngth:= 0;
  buffer:= '';
  int_con_too_long:= false;
  $$
```

Figure 16. Lily code for the language Lily

```
%productions
       {Note that the top three goals share a right paren
        as a prefix.  They are placed at the top to
        facilitate disambiguation.}
  [rt_paren_ast]  : ')','*';
  [rt_paren_plus] : ')','+';
  [rt_paren]      : ')';
  [minus]         : '-';
  [int_con]       : (@intobuf, digit)+, @check_length;
  [term_ident]    : @intobuf, letter, (@intobuf,
    let_num_dash)*;
  [nterm_ident]   : '[', [term_ident], ']';
  [syn_att_ident] : '<', [term_ident];
  [inh_att_ident] : '>', [term_ident];
  [act_sym_ident] : '@', [term_ident];
  [left_paren]    : '(';
  [colon]         : ':';
  [semicolon]     : ';';
  [sheffer]       : '|';
  [comma]         : ',';
  [ellipsis]      : '.', '.';
  [equals]        : '=';
  [character]     : ''', @intobuf, universe, ''';
  [asterisk]      : '*';
  [plus]          : '+';
  [null_string]   : '#';
  [end_of_file]   : 26;
  [keyword]       : '%', [term_ident], @search_keyword;
  [pascal_stuff]  : '$', '$', @read_stuff, '$', '$';

%null: ('{', (comment_char)*,'}') | (' ')+ | 10 | 13;

%aft
  $$
  if return_ = error_ then writeln('Lexical error!');
  $$

%actions
  search_keyword
    $$          (* define return value for non-default goal*)
    return_:= search_key(buffer, error_);
    $$

  intobuf
    $$
    if lngth < max_string then
      begin
      buffer:= buffer + upcase(chr(token_));
      lngth:= lngth + 1;
      end
```

Figure 16. (continued) Lily code for the language Lily

```
    else
      begin
      state_:= accept_;
      return_:= error_;
      end;
    $$

  read_stuff
    $$
    while (look_getchar_l(1) <> 36) or
        (look_getchar_l(2) <> 36) do
      begin                     (* 36 is ASCII
      dummy:= getchar_l;
      if dummy = 26 then        (* 26 is EOF mark *)
        begin
        writeln('EOF encountered in pascal code block');
        exit;
        end;
      end;
    action_:= shift_;$$

  check_length
    $$ if lngth > max_string then return_:= error_; $$

%disambig
  0, 41, 18
    $$          (* disambiguation for top three goals *)
    case look_getchar_l(1) of
    42: state_:= 12;
    43: state_:= 15;
    end;
    action_:= shift_ (*   -2   *);
    $$


%parser lil2pars;
%receives lil2lex;

%local
  $$
  var current_name: charbuf;
  $$
%goals compiler;

%productions

[compiler]: [global_part], [parser_part], end;
[global_part]: global, pascal_stuff | #;
```

Figure 16. (continued) Lily code for the language Lily

```
[parser_part]: (parser, @save_name, term_ident, [att_decl],
  (look, int_con | #), semicolon, [parser_tail])*;
[parser_tail]: @check_getchar, [receives_part], (local,
  pascal_stuff | #), [goals_part], [keywords_part],
  [sets_part],  (fore, pascal_stuff | #),
  [productions_part], [null_stmt],  (aft, pascal_stuff |
  #), [actions_part], [disambig_part] |  @check_name;
  [att_decl]: @check_getchar, left_paren, [att_list_decl],
  (semicolon, [att_list_decl])*, rt_paren | #;
[att_list_decl]: [att_ident_list], colon, term_ident;
[att_ident_list]: [attribute], (comma, [attribute])*;
[attribute]: inh_att_ident | syn_att_ident;

[receives_part]: receives, term_ident, [rec_att_list],
semicolon;
[rec_att_list]: left_paren, term_ident, (comma,
  term_ident)*, rt_paren | #;

[goals_part]: goals, [goal_ident], (comma, [goal_ident])*,
  semicolon;
[goal_ident]: term_ident | left_paren, term_ident, rt_paren;

[keywords_part]: keywords, colon, term_ident, equals,
term_ident, (comma, term_ident)*, semicolon | #;

[sets_part]: sets, ([set_def], semicolon)+ | #;
[set_def]: term_ident, equals, [set_expr];
[set_expr]: [set_term], ((plus | minus), [set_term])*;
[set_term]: [set_factor], (asterisk, [set_factor])*;
[set_factor]: [ordinal_range] | minus, [set_factor] |
  term_ident   | left_paren, [set_expr], rt_paren;
[ordinal_range]: [ordinal], (ellipsis, [ordinal] | #);
[ordinal]: character | int_con;

[productions_part]: productions, (nterm_ident, colon,
  [automaton], semicolon)+;
[automaton]: [auto_term], (sheffer, [auto_term])*;
[auto_term]: [auto_factor], (comma, [auto_factor])*;
[auto_factor]: character | int_con | term_ident |
  nterm_ident |  null_string |  act_sym_ident | left_paren,
  [automaton],  (rt_paren | rt_paren_plus |  rt_paren_ast);

[null_stmt]: null, colon, [automaton], semicolon | #;

[actions_part]: actions, (term_ident, pascal_stuff)+ | #;

[disambig_part]: disambig, (int_con, comma, (int_con |
  null_string), comma, int_con, pascal_stuff)+ | #;
```

Figure 16. (continued) Lily code for the language Lily

```
%actions
  save_name
  $$
  current_name:= buffer;
  $$

  check_name
  $$
  if (current_name <> 'GETCHAR') and (current_name <>
      'GETCHAR_L') then
    begin
    writeln('Semantic Error.');
    writeln('User-defined parsers must have parser tails.');
    return_:= error_;
    state_:= accept_;
    end;
  $$

  check_getchar
  $$
  if (current_name = 'GETCHAR') or
      (current_name = 'GETCHAR_L')     then
    begin
    writeln('Semantic Error.');
    writeln('GETCHAR parsers cannot have attributes or');
    writeln('parser tails.');
    return_:= error_;
    state_:= accept_;
    end;
  $$

%end
```

Figure 16. (continued) Lily code for the language Lily

```
%global  {Program draws graftal plants.}
  $$
{$I graph.p}
  $$
             {request standard parser}
%parser getchar_1;

             {lexical analyzer}
%parser graflex;
%receives getchar_1;
%local
  $$
  type word_type = string[20];
  var buffer: word_type;
  $$
%goals (word);
%keywords: word_type =
  day, night, grass, bush, shoot, leaf, flower, tree, trunk,
  node, branch, cactus;
%sets
  letter = 'a'..'z' + 'A'..'Z';
%fore
  $$
  buffer:= '';      {initialize "local" buffer}
  $$
%productions
  [word]: (@intobuf, letter)+, @search_key;
%null: (' ')+ | 13,10;
%actions
  intobuf
    $$
    buffer:= buffer + upcase(token_);
    $$
  search_key
    $$
    return_:= search_key(buffer, error_);
    $$
             {end of lexical analyzer}

%parser grafpars;
%receives graflex;
%local
  $$
const max_stack = 50;
      stack_pos: integer = 0;
var cor_stack: array[0..max_stack] of record x,y: integer;
end;
    i,j: integer;
```

Figure 17. Lily code for graftal plant language

```
procedure store_loc;
  begin
  stack_pos:= stack_pos + 1;
  with cor_stack[stack_pos] do
    begin
    x:= xcor;
    y:= ycor;
    end;
  end; {store_loc}

procedure recall_loc;
  begin
  if stack_pos = 0 then
    begin
    writeln('Stack underflow');
    exit;
    end
  else
    with cor_stack[stack_pos] do
      begin
      setposition(x, y);
      stack_pos:= stack_pos - 1;
      end;
  end;

function rand_sign: integer;
  begin
  if random > 0.5 then rand_sign:= 1
  else rand_sign:= -1;
  end;
    $$

%goals forest;

%fore
  $$
  graphcolormode;
  palette(0);
  randomize;
  nowrap;
  $$

%productions
[forest]: (day , @draw_day| night, @draw_night),[ground],
  [foliage];
[ground]: # | grass, @draw_grass;
[foliage]: ([bush] | [tree] | [cactus])*;
[bush]: bush, @get_root, [bush_body];
```

Figure 17. (continued) Lily code for graftal plant language

```
[bush_body]: ( @store_loc, (branch, @draw_branch)+, (shoot,
   @draw_shoot)+, @recall_loc )+;
[tree]: tree, @get_root, [tree_body];
[tree_body]: ( (trunk, @draw_trunk)+, [branch_group])*;
[branch_group]:  node, (@store_loc, branch, @draw_branch,
(leaf,   @draw_leaf | [branch_group]), @recall_loc)+;
[cactus]: cactus, @get_root, (@store_loc, branch,
  @draw_limb, (flower, @draw_flower | #), @recall_loc)+;

%aft
  $$
  readln;
  textmode;
  $$

%actions
  draw_leaf
  $$
  i:= random(10) + 5;
  setheading(random(320));
  setpencolor(1);
  forwd(i);
  turnleft(120);
  forwd(i);
  turnleft(120);
  forwd(i);
  turnleft(150);
  penup;
  forwd(i div 2);
  pendown;
  fillshape(xcor + 159, 100 - ycor, 1, 1);
  $$

  draw_trunk
  $$
  setpencolor(2);
  setheading(rand_sign * random(20));
  forwd(random(20) + 20);
  $$

  draw_branch
  $$
  setheading(random(180) - 90);
  setpencolor(2);
  forwd(random(20) + 10);
  $$
```

Figure 17. (continued) Lily code for graftal plant language

```
draw_limb
$$
setheading(random(180) - 90);
setpencolor(1);
forwd(random(30) + 10);
$$

draw_flower
$$
circle(xcor + 159, 100 - ycor, random(5), 3);
fillshape(xcor + 159, 100 - ycor, 3, 3);
$$

draw_shoot
$$
store_loc;
setpencolor(1);
setheading(random(320));
forwd(10);
recall_loc;
$$

draw_day
$$
graphbackground(lightblue);
i:= random(295)- 147;
j:= 60 + random(30);
setposition(i,j);
circle(xcor + 159, 100 - ycor, 25, 3);
fillshape(xcor + 159, 100 - ycor, 3, 3);
$$

draw_night
$$
graphbackground(black);
for i:= 0 to 1000 do
  plot(random(320), random(200), 3);
i:= random(295)- 157;
j:= 60 + random(30);
setposition(i,j);
setheading(45);
arc(xcor + 159, 100 - ycor, 180, 25, 3);
setposition(i,j);
setheading(75);
arc(xcor + 159, 100 - ycor, 110, 30, 3);
setposition(i+5, j+2);
fillshape(xcor + 159, 100- ycor, 3,3);
$$
```

Figure 17. (continued) Lily code for graftal plant language

```
  draw_grass
  $$
  setposition(-159, -99);
  setheading(90);
  setpencolor(1);
  forwd(318);
  turnleft(90);
  forwd(15);
  turnleft(90);
  forwd(318);
  turnleft(90);
  forwd(15);
  fillshape(2, 195, 1, 1);
  $$

  get_root
  $$
  setposition(random(310) - 150, -84);
  $$

  store_loc
  $$
  store_loc;
  $$

  recall_loc
  $$
  recall_loc;
  $$

%end
```

Figure 17. (continued) Lily code for graftal plant language