A FORMALIZATION AND CORRECTNESS PROOF OF THE CGOL LANGUAGE SYSTEM

Michael Lee Van De Vanter

March 1975

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC
CAMBRIDGE MASSACHUSETTS 02139

A FORMALIZATION AND CORRECTNESS PROOF OF THE CGOL LANGUAGE SYSTEM

by

Michael Lee Van De Vanter

Submitted to the Department of Electrical Engineering on
January 22, 1975 in partial fulfillment of the requirements
for the Degree of Master of Science.

ABSTRACT

In many important ways the design and implementation of programming languages are hindered rather than helped by BNF. We present an alternative meta-language based on the work of Pratt which retains much of the effective power of BNF but is more convenient for designer, implementer, and user alike. Its amenability to formal treatment is demonstrated by a rigorous correctness proof of a simple implementation.

THESIS SUPERVISOR: Vaughan R. Pratt
TITLE: Assistant Professor of Computer Science and Engineering

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Vaughan Pratt, who provided the inspiration and support for this work; to Yuval Peduel who offered tireless criticism and encouragement; to Donna Jean Brown who helped make this thesis comprehensible; to Al Nemeth who made helpful comments on the final draft; and finally to Cleavon who was always willing to listen.

This research was supported by the National Science Foundation under research grant nos. GJ-34671 and GJ-43634X

TABLE OF CONTENTS

I. INTRODUCTION

The design and implementation of programming languages is a complex problem which must be addressed from at least four distinct viewpoints. These viewpoints reflect the different but interacting interests of the designer, implementer, user, and theoretician. We address specifically the kinds of problems evident in the following two scenarios:

Scenario 1: The old dangling ELSE problem.

> An early ALGOL grammar in Backus Nauer Form (BNF) was ambiguous with respect to nested IF-THEN-ELSE statements. This was noticed by implementers who often adopted the fairly local solution of attaching an ELSE to the most recent available THEN. Although BNF grammars were eventually discovered corresponding to this resolution, the grammar for ALGOL was rewritten to simply forbid nested conditionals [Nauer 1963].

Scenario 2: A new, theoretically sound approach.

> This is a summary of advice given for the construction of deterministic parsers and translators in *The Theory of Parsing, Translation and Compiling* [Aho & Ullman 1972].
>
> 1) Write your grammar in BNF.
>
> 2) Decide whether you want top-down or bottom-up parsing (top down is more flexible for translation).
>
> 3a) If you choose top-down: apply known transformations to the grammar and check the result for the LL(1) property. If successful, a reliable top-down parser may automatically be constructed which handles a general class of syntax directed translation.

3b) If you choose bottom-up: attempt to modify the grammar to satisfy the SLR(1) or LALR(1) conditions. If successful, a bottom-up parser may be likewise constructed.

4) In both cases, especially bottom-up, apply known optimizing transformations to the parsers to attain practical efficiency.

In the first scenario BNF is being used as the formal reference tool, since it enables precise syntactic description. It does not, however, reveal important properties (e.g. ambiguity) which the language designer needs to consider. Further, the implementer must work informally, since the grammar itself does not suggest efficient parsing techniques (see the survey of various approaches in *ALGOL 60 Implementation* [Randell 1964]). Finally, evidence indicates that the user may also be using informal syntactic models (see the description of expression evaluation in *Introduction to ALGOL* [Baumann 1964]). This situation precludes any serious attempt at formal verification.

A considerable amount of rigor has been obtained via the formal approach in the second scenario, but Aho and Ullman acknowledge several shortcomings. Many grammars cannot be made LL(1) and, even when they can, the resulting grammars are usually large and awkward and thus unnatural for syntax directed translation rules. Formal techniques do not exist for obtaining SLR or LALR grammars. Finally, in both cases nontrivial changes to the original grammar usually require that the entire process be repeated.

A fundamental weakness with these approaches is that BNF is inappropriate as a definitional meta-language; it is essentially based on theories of generative grammars. The practical demands of parsing and translating restrict us to certain "tractable" grammars, but such grammars are often very difficult to recognize. In addition these "tractable" grammars tend not to include the most convenient description of a language, so one usually ends up with several representations for the same language definition; e.g., a simple grammar for the user, and a complicated one for the parser. Finally, it is often necessary to transform the grammar into a parse table and then into an optimized parse table. Such multiple representations form a severe obstacle to formal verification.

What we would like, then, is a system which includes:

1) A natural and convenient definitional meta-language for the designer,

2) A user oriented meta-language which makes any defined language easy to learn and use,

3) A simple method for automatically constructing an efficient parser/translator for any defined language, and

4) Enough precision in the above to permit formal proof that all components agree precisely.

Pratt presents a system in "Top Down Operator Precedence" [Pratt 1973] which addresses the first three of these issues quite well. He allows the implementer to "write arbitrary programs" while offering "in place of the rigid structure of a BNF-oriented meta-language a modicum of supporting software, and a set of guidelines on how to write modular, efficient, compact and comprehensible translators while preserving the impression that one is really writing a grammar rather than a program." This approach has been followed in the construction of CGOL, a combination definitional meta-language and extensible programming language [Pratt 1974] which demonstrates the power and convenience inherent in this approach.

The CGOL system, as presented, does not satisfy the fourth criterion; it lacks a complete formal context in which correctness may be stated and proven. In this paper we complete a formal context, present an example implementation, and rigorously prove its correctness.

We believe that many of the difficulties mentioned above may be avoided by writing grammars in a meta-language whose descriptive power is tailored to fit the intended application. We present and analyze such a meta-language for CGOL type translation; the meta-language expresses a class of languages which are easily and naturally parsed. For an exact definition of the describable languages, we present a user-oriented model which describes how sentences may be generated from any grammar.

Since the meta-language is designed to fit the parsing method, it is possible to construct an extremely simple parsing program which operates by simply reading a given grammar as data. We give a LISP implementation of this parser, designed primarily for clarity and ease of proof.

The correctness proof for the example parser is presented in two parts; theoretical properties and a program proof. The theorems of the first part deal exclusively with properties of the meta-language; these proofs are completely independent of the program and the parsing algorithm. The use of these properties allows the actual program proof to deal almost exclusively with argument passing and flow of control; the program proof is tedious but straightforward.

Chapter II contains an introduction and analysis of the CGOL approach to parsing. Chapter III is an introduction and informal discussion of our system: the syntactic meta-language, the generative model of defined languages, the parsing program, and correctness criteria. Chapter IV covers the same material with complete formal definitions, and Chapter V contains the correctness proof

II. THE CGOL APPROACH

We begin with a presentation and analysis of the parsing/translating method proposed by Pratt; a motivation and detailed introduction may be found in "Top Down Operator Precedence" [Pratt 1973]. The discussion in this chapter centers on the parsing technique: how it works, what features yield unique advantages, and how it compares with known work in formal parsing theory.

II.A The Algorithm

Pratt's approach to translation (which we refer to as the CGOL approach, after its application in [Pratt 1974]), is specifically oriented toward the translation of *expressions*, where an expression is simply an operator (e.g., + or ∗) with its arguments. For those not familiar with expression oriented programming languages, the analogy to arithmetic expressions is sufficient for the moment. Each operator of the defined language has associated with it a program which embodies most syntactic and semantic information for that operator. The programs, called *denotations*, are executed in a left to right scan by a simple, recursive algorithm; each denotation has the power to look at the next symbol in the input string, advance (but not back up) the current symbol pointer, and call the parsing algorithm recursively to scan another expression. The pointer to the input string is a global variable and may be advanced by any denotation. The denotation of a symbol may be called at two points in the algorithm: step 2 and step 4. Step 2 corresponds to the case where the operator is at the beginning of a string and does not take a left argument. Step 4 assumes that the expression parsed so far is the left argument to the operator.

PARSE is the function which is called to scan and translate an expression starting at the beginning of an input string.

STEP 1: PARSE looks at the first symbol of its input string (it will never look farther ahead than the current pointer to the string). Since this symbol occurs at the beginning of an expression, it is assumed to be an operator which takes no argument on its left side (constants and variables are treated as operators with *no* arguments). PARSE executes the denotation associated with this symbol.

STEP 2: The denotation for the current operator moves the pointer rightward along the input string, when necessary to gather right arguments. The denotation returns the translation of this expression, leaving the input pointer at the symbol following the expression.

STEP 3: PARSE now has the translation of an expression starting at the beginning of the input string. The question is asked: should this expression be given as a left argument to the next operator in the string, or should it be returned (presumably as a right argument to the caller of PARSE)? The decision is made by comparing numerical binding powers associated with each operator; the next symbol must have a *left binding power* associated with it, and PARSE was given, as an argument, the *right binding power* of its caller.

STEP 4a: If the right binding power of the caller is greater (or equal), the translation obtained so far is returned. RETURN.

STEP 4b: If the left binding power of the next symbol is greater, then it is assumed to be an operator, and the expression translated so far is its left argument. The translation is passed as an argument to the execution of the denotation associated with this symbol.

STEP 5: The denotation for this operator moves the pointer rightward along the input string, when necessary to gather right arguments. The denotation returns the translation of this expression, leaving the input pointer at the symbol following the expression.

STEP 6: Iterate to step 3.

We observe that the definitional information for each operator falls into four general categories. In the first category we include the specification of the operator's left and right binding powers; these integers are used to locate the right ends of expressions. The second category simply indicates the presence or absence of a left argument. This feature belongs in a separate category since the collection of a left argument is not directly controlled by a denotation;i.e., when a denotation is executed, its left argument, if any, has already been scanned and translated. Denotation for operators with a left argument are executed from Step 4b, those without from Step 2. The third category includes a procedure for right argument collection which may invoke a number of techniques, the most obvious of which is the collection of an expression (argument) by recursively calling the parser. In addition, parsing decisions may be made by looking one symbol ahead in the input string. The fourth category includes a procedure for translation.

II.B Comparisons with other Methods

From a theoretical standpoint a CGOL translator has unlimited syntactic power. This is not, however, the primary issue; it is much more important to ask what it can do *well*. We provide one answer to this question by comparing the algorithm to a number of known parsing methods, showing how CGOL combines certain advantages of each. This discussion presupposes some familiarity with formal parsing theory. The topics discussed are:

1. Introduction and Example Grammars
2. The Parse Type
3. Skeletal Grammars, Ambiguity
4. Operator Languages, Precedence Parsing
5. Flow of Control
6. Combination Unary/Binary Operators

1. Introduction and Example Grammars

The key to the effectiveness of the CGOL parser is the simple but powerful control structure. The syntactic power of the parser is theoretically unlimited, since arbitrary programs may be written as denotations; the control structure, however, creates an environment in which a great many grammatical constructs may be handled very simply

The language CGOL presented in [Pratt 1974] and the translator constructor in this paper are examples. This flexibility and convenience result from a unique combination of parsing techniques, most of them well known by themselves. Rather than asking to which theoretical class CGOL belongs, we look for similarities between the operation of the CGOL parser and those in known categories. CGOL combines advantages from many different approaches.

We will refer to the following grammars in this discussion. They illustrate in a simple way several of the issues relevant to parsing schemes. Example A is an ambiguous grammar for the language of arithmetic expressions; A' is a standard unambiguous version in which + and $*$ associate to the left and ↑ associates to the right. These properties result from the use of single productions and left and right recursion. B is an ambiguous grammar for IF-THEN-ELSE statements (the well-known dangling ELSE problem). Grammar B' is an unambiguous grammar for the same language, representing the usual solution to the problem.

Grammar A

1 $E \rightarrow E + E$

2 $E \rightarrow E * E$

3 $E \rightarrow E \uparrow E.$

4 $E \rightarrow (E)$

5 $E \rightarrow a$

Grammar A'

1 $E \rightarrow E + T$

2 $E \rightarrow T$

3 $T \rightarrow T * F$

4 $T \rightarrow F$

5 $F \rightarrow P \uparrow F$

6 $F \rightarrow P$

7 $P \rightarrow (E)$

8 $P \rightarrow a$

Grammar B

1 S → if B then S
2 S → if B then S else S
3 S → c
4 B → b or B
5 B → b

Grammar B'

1 S → if B then S
2 S → if B then S' else S
3 S → c
4 S'→ if B then S' else S'
5 S'→ c
6 B → b or B
7 B → b

2. The Parse Type

If we trace the operation of the CGOL parser, observing the order in which the components of the parse tree are recognized and assembled, we see that it is essentially producing a left corner (LC) parse. We begin the discussion of this observation with a brief look at top-down parsing. Parse types are categorized top-down, bottom-up, etc. according to the order in which they recognize the grammar rules used to derive the input sentence. An equivalent model is to imagine the derivation as a tree with the root nonterminal symbol at the top, and the leaves corresponding to the sentence. A top-down parser recreates this tree from the top downward, root nonterminal first. Stearns points out that this type of parser is especially useful for combined parser/translators; since each production is identified before its descendents in the tree, an implementation may conveniently use recursive descent. Translation rules may correspond to grammar rules, which may correspond to nested environments in the translating program. These ideas are discussed at length in [Knuth 1968] and [Lewis & Stearns 1968].

LL Languages

The LL(k) grammars are those which can be naturally parsed deterministically (i.e., without backtrack as the input is scanned) from left to right, top-down. The usual parser associated with LL grammars is the predictive parser which looks ahead k symbols on the input stream before deciding which production to recognize at any given point in the parse. In addition to the general usefulness of top-down parsing, predictive parsers for LL(k) grammars are very simple; they may be implemented on a one-state Deterministic Push Down Automaton (DPDA) [Kurki-Suonio 1969]. Further, they are very efficient and handle errors reasonably well [Aho & Ullman 1972].

The central problem with LL parsing is that very few grammars are LL(k). In fact, very few languages have LL(k) grammars for any k; an example is grammar B', which generates a non-LL language. When languages do have LL(k) grammars, these are not always the smallest or most natural descriptions of the language. For example, Stearns discusses transformations which may convert grammars into LL(1) grammars at the expense of added complexity [Stearns 1971]. Grammar A' for arithmetic expressions is not an LL grammar for any k because of left recursion (in rules like $E \rightarrow E + T$). Left recursion may be eliminated by converting a grammar to Greibach Normal Form (via a known algorithm). The GNF grammar for arithmetic expressions is essentially right associative, although the old grammar parse may always be recovered from a new grammar parse. Stearns presents optimizations which reduce the nonterminal explosion in the case of arithmetic expressions (in general the transformation squares the number of nonterminal symbols), but the result depends heavily on the fact that this is an operator precedence language. This property of arithmetic expression grammars (such as A') allows a simpler treatment by the direct use of operator precedence (to be discussed below).

Left Corner Parsing

As mentioned above, LC parsing is a variant on top down parsing. While a top down parser must recognize the occurrence of a rule before any of its descendants, an LC parser does not until the leftmost descendant has been found. This leftmost descendant, the leftmost symbol in the right part of the rule, is called the left corner. This corresponds quite closely to the operation of the CGOL parser; each rule in CGOL corresponds to an operator, and each operator is recognized (its denotation executed) as it is encountered in a left to right scan. Since operators may have expressions occurring as left arguments, they are recognized after their left corner. This parse method has been said to parse the left corner of a rule bottom-up and the rest of the rule top-down. When the first symbol of a rule is a nonterminal symbol, as with all NILFIX and PREFIX operators in CGOL, the parser is operating essentially top down.

Nondeterministic LC parsing has been used for some time [Irons 1961] [Cheatham 1967], but only more recent work has examined deterministic LC parsing. Rosenkrantz and Lewis identify the LC(k) languages, those which have LC(k) grammars and can be parsed deterministically LC with k symbol lookahead [Rosenkrantz & Lewis 1970]. The class of LC(k) languages is shown to be identical to the class of LL(k) languages via the result that the elimination of left recursion produces an LL(k) grammar if and only if the original grammar was LC(k). Thus LC(k) grammars give us no ultimate increase in expressive power, but they do offer a naturalness and economy of description in many cases. In an LC(k) translator this advantage is gained at the cost of some potential flexibility (since left corner nonterminals may not be parsed top-down). An important advantage is that one rule corresponds to one operator, and the semantics for a rule may be conveniently localized.

Grammar A' is LC(1), and thus a transformed version, without left recursion, is LL(1); in fact, this is nearly identical to the example transformed by Stearns in [Stearns 1971] where the number of nonterminals becomes squared under the transformation. Grammar B', however, is not LL(k) for any k. In fact, it is intuitively clear that the language generated by B' is not an LL language, since it is impossible to tell at the begining of a string which of two rules is to be applied; there can be no LL(k) or LC(k) grammar which generates the language.

3. Skeletal Grammars

While the CGOL parser traces a left corner parse and operates with lookahead 1, it is not actually an LC parser as defined by Rosenkrantz and Lewis, since it uses no grammar in the ordinary sense. There is only one nonterminal in the parser, the implicit one for an expression. All expressions are treated the same. What we have then is more like the grammar A', sometimes called a skeletal grammar. Skeletal grammars typically are ambiguous, so external means need to be used to resolve any ambiguous sentences. The CGOL parser resolves this ambiguity by a number of techniques sometimes seen in parser implementations, linear operator precedence functions, flow of control decisions, and two-state unary vs. binary operator recognition. Some of these techniques have been viewed as optimizations to be used whenever a grammar is found with the right property, although it is seldom obvious at a glance if this is the case. Techniques have even been developed to transform grammars in the hope that the desirable properties might be obtained.

The CGOL approach is to avoid juggling context-free grammars at all. This is done by not attempting to describe difficult matters with cfg rules. These rules are certainly useful for describing phrase structure (as in the two ambiguous example grammars), but begin to grow in size and lose clarity when they describe operator hierarchies and association (as in grammar A).

4. Operator Languages, Precedence Parsing

Some of the information which is normally represented by nonterminal symbols may be defined as properties of the terminal symbols, if the languages are defined by operator grammars. These are context-free grammars which have no adjacent nonterminal symbols. Although these are something of a special case in the literature on formal languages, a great many useful programming languages have (or are very close to having) operator grammars. All four example grammars are operator grammars; see also [Floyd 1963] for an operator grammar for ALGOL. In fact, it seems that adjacent nonterminals usually appear when we try to solve some "problem" with a grammar (say ambiguity, or left recursion) by transforming it into something less natural. Rules with no nonterminal symbols at all are especially nonintuitive; we like to think of each rule as having some meaning, but when a rule has no associated terminal symbols, its occurrence relative to a sentence will only be implicit. In the CGOL parser each rule is attached to some symbol, an operator. With this restriction CGOL is able to apply the following techniques.

Precedence Parsing

The term precedence parsing describes a well known family of techniques used in bottom-up parsing. The standard implementation of a bottom-up parse is known as a shift-reduce algorithm. This algorithm scans the input, one symbol at a time, from left to right. A shift step reads an input symbol and pushes into onto a stack. A reduce step occurs when a sequence of symbols on the top of the stack correspond to the right side of a grammar production; this leftmost reducible phrase is called the "handle" of a sentential form. This series of symbols is popped off the stack and is replaced by the nonterminal symbol on the left of the rule. A parse is complete when the stack contains only the root nonterminal of the language and the input stream is empty; the output is a bottom-up parse.

Precedence parsing methods are distinguished by the method of making the shift-reduce decision, i.e. deciding if the scan has reached the right end of a handle. The general technique is to derive from the grammar a relation (usually written ⋗) on the symbols of the language. Although a variety of precedence techniques have been developed, their essential feature is that they compare two adjacent symbols in a sentential form; if the relationship ⋗ holds between them, the right end of a handle has been reached.

Operator Precedence

The application of precedence techniques to operator languages leads to a well known and efficient parsing method (see [Floyd 1963]). Operator precedence grammars are those for which the shift-reduce decision may be made uniquely by considering only terminal symbols; i.e., the uppermost terminal symbol on the stack is compared with the next input symbol. Considerable storage space and algorithmic complexity is saved by simply ignoring nonterminal symbols; i.e., not using them to carry information. The resulting parse tree is called the skeletal parse, since all productions with single nonterminals on the right side are missing. The interesting structure is there, though, since extra nonterminals with rules like E → T in Grammar A' are often included only to express properties like right or left association and have no semantic implications.

Although operator precedence seems a somewhat obscure property for a grammar to have, Floyd argues that many useful programming languages are quite close to having operator precedence grammars. He offers an ALGOL operator precedence grammar as an example and identifies certain problems which he suggests be solved via escape clauses, or special parse techniques. It seems that the technique handles the majority of language features quite well, but has certain difficulties which would be much better dealt with by exception, than forced in the basic scheme. CGOL deals with some of these problems quite well.

Pratt conjectures that operator precedence techniques are widely applicable because of their intuitive appeal; they correspond exactly to the ordinary conventions for writing arithmetic operators. Grammar A' for example is an operator grammar in which the relations $\uparrow \gtrdot * \gtrdot +$ hold. These represent the notion of the precedence hierarchy of these operators. We also note that $+ \gtrdot +$ and $* \gtrdot *$, meaning that these two operators associate to the left. On the other hand, the relation $\uparrow \lessdot \uparrow$ holds; this means, in the operator precedence scheme, that this operator associates to the right.

Linear Precedence Functions

An optimization often considered for operator precedence schemes (and for precedence relations in general) is the encoding of the precedence matrix (i.e. the relation) via linear functions. Typically, two integer valued functions f and g are defined over terminal symbols. If for two terminal symbols x and y the relation $x \gtrdot y$ holds, then it will also be true that $f(x) > g(y)$. While the technique only works for a small number of possible matrices, it turns out to be easily applicable to grammars like A'. Again, the conventional hierarchy of the operators in arithmetic expressions allows this encoding scheme to work.

An operator precedence parser for arithmetic expressions is very compact and efficient. CGOL makes use of the operator precedence technique, but without forcing the designer to express his ideas in BNF first, only to have them transformed by algorithm into what might essentially be the original idea. The designer simply defines left and right binding powers for each operator.

We recall that left corner parsing treats the left argument to an operator in bottom-up mode, and the rest of the rule in top-down mode. It is in the bottom-up mode that this technique is used by CGOL. When PARSE has scanned a complete expression, a decision is made by binding powers. If the next token of the string wins the expression, then the expression becomes a left argument. If PARSE returns the expression, then the expression is the result of a top-down call from some higher level. The operation of CGOL for grammars composed of only arithmetic operators, like 0, is exactly parallel to the operation of the canonical strong LC machine of Rosenkrantz and Lewis [Rosenkrantz & Lewis 1970]. The nested environments of CGOL correspond to the stack of the LC machine. An LC stack entry may either be a single nonterminal symbol, corresponding to a call to PARSE which has not yet parsed an expression, or a pair of nonterminal symbols, corresponding to a call to ASSOC which already has a left argument (or left corner) parsed, waiting to be attached to something.

5. Flow of Control

A major difference between CGOL and the LC machine becomes clear when we consider grammar B'. This is an operator precedence grammar which is easily handled by traditional bottom-up methods, but it is not LL(k) for any k. By the result of Rosenkrantz and Lewis then it is also not LC(k) for any k. The CGOL parser handles this example with great ease, since the program for the operator IF can simply parse its THEN argument and then look one token ahead to see if it is ELSE. Both possibilities are treated by the same denotation, so we are using the equivalent of the ambiguous version, grammar B. As with arithmetic expressions, CGOL uses an ambiguous grammar with a simple rule to resolve ambiguity; in this case it is simply to take the ELSE if it is there. Aho, Johnson, and Ullman treat this example in some detail, pointing out that this solution is a simple fix to the otherwise ambiguous top-down parsing table for grammar B [Aho, Johnson, & Ullman 1973]. We have a situation where the top-down predictive parsing technique works for cases which are outside of the normally defined LL boundaries. By allowing arbitrary programs as denotations, CGOL allows an operator to collect any right arguments in a very general top-down fashion. We might say that each operator has its own top-down predictive parser for the grammar of its right arguments. It is this feature which allows the use of regular expressions to specify annotation patterns within the meta-language defined in this paper. In fact the restrictions placed on the use of the regular operators make each annotation pattern the equivalent of a miniature LL(1) language, although the restrictions are in fact even stronger than LL(1).

6. Combination Unary/Binary Operators

The third technique used to resolve ambiguity in a CGOL parser is a solution to a problem encountered by Floyd when he tried to write an operator precedence grammar for ALGOL. Certain symbols of the language have two uses, and operator precedence by itself can not distinguish between them. The common example of this is the minus operator which may be used either as unary or binary. CGOL allows this double definition in a general form. Any operator may have two unrelated definitions if one of them has a left argument and one does not. CGOL is in this sense a two state machine, one state corresponding to an immediate call to PARSE, when no left argument is present, and the other to a call to ASSOC, when there is a left argument available. There is never any ambiguity.

III. BASIC CONCEPTS

In this chapter we motivate and informally introduce the components of our language system. The notions presented will be given full formal treatment in the following chapters. We discuss first the meta-language, giving examples of its use. Since the meta-language is nonstandard, we will present a generative model which determines the sentences of a defined language. The chapter concludes with a brief discussion of the translator algorithm and its correctness criteria.

III.A The Meta-Language

Our formal language system is based on a syntactic meta-language which:

(a) restricts the syntactic power of the system in a way which permits rigorous proofs,

(b) embodies the full power of the scheme in the sense that we want it to express anything which the parse/translation scheme handles naturally and efficiently, and

(c) allows the automatic construction of simple translators.

We recall from Chapter II that the translator uses four types of information for each operator in the defined language:

(1) Left and right binding powers,

(2) Presence of left argument,

(3) Pattern of right and annotated arguments, and

(4) Translation rule

In the original CGOL facility this information is specified by the designer in a varying mixture of declarative and procedural modes. To facilitate uniform treatment, we will allow exactly one type of meta-language statement, a *production*, which will contain all of the data necessary to define a single operator. We restrict the syntactic power of the translator by requiring that all syntactic information (parts 1-9 as listed above) be stated in a declarative language, leaving only the translation rule in procedural form. This declarative segment includes a template of argument positions (parts 2 and 3) and a specification of binding powers (part 1). Thus we might write:

Ex. 1 ~ "+" ~ ,14,14;<denotation>

to define + as an operator of the language with left and right arguments. It has left and right binding powers of 14, and <denotation> is a procedure which accepts as input the translations of the arguments and calculates the translation of the entire phrase. To deal with more general programming language features we allow productions like:

Ex. 2 "IF" ~ "THEN" ~ ("ELSE" ~ | λ) ,6;<denotation>

which defines the standard conditional operator. This production includes the specification of extra right arguments (in addition to the normal one with IF acting as a prefix operator); we call "THEN" ~ ("ELSE" ~ | λ) the *annotation pattern* of the operator "IF". Here the alternation (or union) symbol | is used to specify a choice of two patterns, one of which is the null string λ. An even more powerful conditional may be specified by the production:

Ex. 3 "IF" ~ "THEN" ~ ("ELSEIF" ~ "THEN" ~)* ("ELSE" ~ | λ) ,6;<den>

which uses the star closure symbol * to indicate any number of occurrences.

We write annotation patterns using regular expression notation (as in Examples 2 and 3) because it is well known, quite general, and amenable to formal treatment. In an actual implementation one might extend this notation to include pattern operations expressible in terms of the basic notation. For example, we might introduce the brackets [and], and let [a] denote (a|λ). We could then write simply:

```
Ex. 2'          "IF" ~ "THEN" ~ ["ELSE" ~] , 6;<denotation>
```

instead of Example 2. Another possibility might be < and > to mean + closure (one or more occurrences). Such extensions are not included here, since they do not affect the theoretical behavior of the meta-language.

This meta-language is restricted enough to allow formal treatment (goal a above) and is general enough to exploit the power of the parsing scheme (goal b). The patterns, however, are too powerful for simple parsing (goal c); any of these patterns could theoretically be parsed, but not all of them easily or unambiguously. We solve this by restricting the class of permissible patterns to those within the power of a very simple parsing algorithm.

This matching algorithm for patterns (arguments on the right side of operators) is deterministic and never looks more than one symbol ahead in the input string. Our model of the algorithm is a person with one finger on the pattern, one finger on the input string, and almost no memory! It should always be clear what to do next; no backing up allowed. To put this differently, the user should always be able to understand the parsing method. To insure the correct operation of the parser, we adopt the following three rules.

The first rule is that patterns joined by alternation not begin with the same symbol. Thus we disallow the pattern:

```
Ex. 1'     "IF" ~ ("THEN" ~ | "THEN" ~ "ELSE" ~) ,6;<denotation>
```

as an alternative to Example 1. In fact we prefer the original form for the following reason: an annotated argument should be identified by the name of the preceding symbol, not by its position in a pattern. We intend that there be no difference between the two THEN arguments specified in example 1'.

The second rule solves a problem arising from the use of the symbol λ in patterns. Whenever the pattern λ is an alternative, the scanner could "match" λ and miss a non-null matching symbol. This problem is solved by a fiat similar to the dangling ELSE solution. The parser will always match as much of the input string as possible; the pattern λ is always the lowest priority choice.

The third rule prohibits certain other patterns which cannot be completely handled by the parser. For example, we consider the production

Ex. 4 "FOO" ~ ("BAR" | λ) "BAR" ,2;<denotation>

which describes two possible phrases; one has one occurrence of BAR, the other has two. Because of the fiat above, our algorithm can only parse the second possibility correctly. This a local case of the dangling ELSE problem, and since it is detectable, we disallow patterns in which it occurs. Informally, this rule restricts the use of patterns which give the parser a choice whether to continue, based on the presence of some delimiter symbol like ELSE. We will require that such a pattern not be concatenated on the left with a pattern which can start with one of its delimiter symbols. In place of Example 4 we might use the production

Ex. 4′ "FOO" ~ "BAR" ("BAR" | λ) ,2;<denotation>

which matches the same phrases but can be parsed correctly.

While not immediately obvious, these restrictions are completely local to each production and are intuitively motivated Patterns which violate them and can sometimes be rewritten in an acceptable form, and the acceptable form often makes more sense. In fact, the verification of these conditions is computationally quite simple and an interactive definitional facility would have verification and debugging aids built-in. These rules are considerably simpler and more intuitively appealing than the LL and LR conditions.

On a global level, use of the meta-language is quite straightforward; the global restrictions which do exist are very simple. Only one production may be given per operator, although some symbols may be used for two different operators, one with a left argument and one without (e.g., the binary and unary minus operators would be defined in two separate productions). A symbol defined as an operator may also be used as a delimiter (in annotation patterns) as long as its binding powers remain well defined, since the role of a delimiter is passive. This sort of detail is trivially manageable by a definitional facility.

An important property of this meta-language is that a set of productions forms a complete language definition; no other information is necessary. It is precisely this extreme modularity which makes designing extensible languages convenient.

III.B User Model

Once we have a language definition, a set of productions, we want to offer the user a manual explaining how to use the defined language. We claim that the productions themselves are straightforward enough (and their syntactic interactions simple enough) to serve as the basic manual, once our generative model is understood. For precision and verification this model will be presented in formal terms. It should be understood, however, that the formalism is intended only to add rigor to intuition; intuition need not be bent in order to agree with formalism. Some of the assumptions on which the model is based are discussed in *Top Down Operator Precedence* [Pratt 73].

The operator is the basic definitional unit in these languages; appropriately, the user's primitive concept is the relation "is an argument of". We carry this one step further by specifying what kind of argument (what role it plays). Also, to allow more than one argument of the same kind, we specify an ordering. It is then natural to represent expressions as trees: nodes correspond to operators and subtrees correspond to arguments. The branches are ordered and labelled to identify the argument: normal arithmetic arguments are connected by branches labelled left or right, and annotated arguments are labelled by the annotating token itself, the delimiter. This is very closely related to McCarthy's abstract syntax [McCarthy 1963].

The purpose of syntactic convention is to uniquely represent these expression trees as linear strings of symbols. Two well known examples are the use of postfix and prefix notation to represent ordered trees. In the domain of binary trees infix notation is commonly used, but here additional conventions are necessary to resolve the association of intervening arguments. An example of this problem is the string a+b*c, where we know by convention that b is the left argument of the operator * and not the right argument of +. The convention used here is usually viewed as a hierarchy of the arithmetic operators in which the higher operators "go first" or "take precedence" over lower operators. We use this convention to recover the correct tree from a given string; it may also be used to determine which trees are directly expressible as strings, and which trees require the use of parentheses.

The languages we define use a combination of notational conventions including infix. To deal with association problems we adopt a convention based on the idea of operator hierarchy. A *binding power* is a numerical value which represents the precedence level of an operator; thus an expression between two operators is understood to be an argument of the operator with the higher binding power. This convention is generalized somewhat by allowing separately specified *left* and *right binding powers* for each operator, allowing operators to behave asymmetrically.

We incorporate this convention in a model for writing linear expressions from trees. The basic rule for writing expressions is: don't use an expression *e* as a left (right) argument to an operator *op* if the left (right) binding power of *op* is high enough to cause any subexpression of *e* to associate incorrectly. We know that a*b may be used as an argument to +, but a+b may not be used as an argument to *. Formally, we measure the resistance (on each side) of an expression to false associations. We will define the r-index (l-index) of an expression to be essentially the lowest right (left) binding power of any internal operator exposed to the right (left) side of the expression. Fo: example, the r-index of a+b*c is equal to the right binding power of +, since an operator to the right of this expression (say another *) might take b*c incorrectly as a left argument. The l-index of the expression SIN a is ∞, since it is totally invulnerable to false associations on the left. Although this model does not allow certain expressions trees to be written, most defined languages include a bracketing operator (like parentheses) which is semantically null and creates an expression with l-index=r-index=0. Thus, (a+b) may be used as an argument to *.

The only other way in which operators may syntactically interact results from the generalized dangling ELSE problem. The expression IF a THEN b has the property that an ELSE occurring immediately after b will cause the parser to continue collecting arguments for this phrase (recall the fiat: given the choice of continuing or not, the parser will always continue). The informal rule is: don't follow an expression e by a delimiter which will get incorrectly included with e (or some subexpression of e). This rule prohibits the use of an IF-THEN expression as the second argument (i.e., the THEN argument) to an IF-THEN-ELSE expression. We formalize this rule by defining the c-set for each expression, the set of tokens which would cause argument collection to continue incorrectly at some level. We say: an expression may not be followed by a token in its c-set.

These three properties (l-index, r-index, and c-set) completely describe the syntactic behavior of any expression. A standard BNF grammar would represent the same information implicitly by the use of one nonterminal symbol. More closely related techniques have been studied which attach various modifiers to nonterminal symbols in context-free grammars; see especially "Indexed Grammars" [Aho 1968] and the transformation defined on well-chained grammars in [Stearns 1971]. The CGOL approach is extreme in the sense that nonterminal symbols play virtually no role at all.

The separate treatment of syntactic properties is an important feature of this approach; both designer and user can deal with the various syntactic issues explicitly and separately. The most prominent syntactic feature of a language is its basic phrase structure, expressed by the productions as an ambiguous context-free grammar with one nonterminal symbol (called "expression"). Argument association is dealt with separately by binding powers, similar to the arithmetic conventions. Pratt argues that binding powers may be usefully assigned on the basis of an implicit hierarchy of data types, corresponding closely to ordinary intuition and conventions for programming languages. The annotation patterns are also treated separately. Delimiters like ELSE which can cause problems can be explicitly noted (an easily computable property) and the operator combinations which interact can be listed. For example, it would be observed that IF-THEN-ELSE expressions interact with themselves if improperly nested. In a well designed language, these interactions will be rather limited in number, freeing the user from this concern in most cases.

III.C Automatic Parsing

Our meta-language defines a class of programming languages for which the CGOL translation technique is particularly appropriate. We demonstrate by presenting a simple parsing program which, when given a set of productions as data, correctly parses sentences of the defined language and can be easily extended to handle translation via denotation programs. The program is a working (although inefficient) LISP implementation which requires the transformation of productions into a suitable LISP representation.

A definitional facility would be a set of programs to provide this and other services to the designer. The *meta-language processor* is a program which accepts productions of the meta-language, either incrementally or in batches, and stores the information. In this implementation the data are simply attached to the name of the operator being defined (via the property list). A facility could also include automatic verification of annotation patterns with debugging advice, and automatic documentation.

Incremental implementations would be convenient and could even be performed on-line. An extreme example is a bootstrap, in which denotations may be written in the language defined so far (e.g. the language CGOL [Pratt 1974]).

III.D Correctness

We consider a formal proof of correctness an essential, practical component of the system; it is pointless to have automatic parsing without a guarantee that no mistakes will be made. The claim we want proven, then, is simple: given any language definition in the proper representation, the parser works correctly.

To say that the parser works correctly requires a precise definition of what it should do. Our specifications of a meta-language and user model provide a formal context in which correctness may be rigorously defined.

We say that a parser operating on some language definition is correct when the following are true:

I. If the expression (i.e., tree) e is written according to convention as the string ω, then the parser will recover the tree e.

II. If the parser recovers a tree e, then the input string is in the defined language.

Part I guarantees that any valid string of the language will be parsed correctly; part II assures that no incorrect strings will be parsed.

The correctness theorem is actually a statement relating the behavior of two functions: writing (mapping trees into strings) and parsing (mapping strings into trees). Both parts of the theorem are proven by induction, but over different domains: part I over the domain of trees, and part II over strings. It is a corollary of the theorem that the languages defined are unambiguous; i.e., no string can be written from more than one tree.

From the standpoint of formal language theory, the theorem is a proof of equivalence of two alternate language definition mechanisms. A *generative* description is presented as the user model; an *analytic* description is implicit in the parsing program.

The proof itself is carried out in two phases. In the first, we prove a number of theoretical properties of the language class, i.e. of the definitional mechanism. These properties are independent of any program or parsing algorithm. Given these results, the actual proof of the parsing program is tedious, but quite straightforward

IV. FORMAL DEFINITIONS

In this chapter we present the formal details of the language system introduced in Chapter III. Section IV.A presents the meta-language; the parsing program for defined languages is given in Section IV.C. The generative model of defined languages, given in Section IV.B, permits a formal statement of parser correctness, discussed and proven in the next chapter.

IV.A The Meta-language

We begin by naming the basic lexical units of our defined languages.

Definition: A token is a single lexical symbol in a defined language.

Notation:

(i) Actual tokens will be represented using only upper case letters; e.g. IF, ELSE, +, and (.

(ii) Lower case letters are used for meta-variables in this discussion; e.g. t (possibly subscripted) refers to some token.

(iii) Greek letters represent strings of tokens; e.g. α, β, γ.

While the token is a lexical unit, the operator is our basic definitional unit.

Definition: An operator is a set of semantic and syntactic information, representing some operation. We use the meta-variable *op* for operators.

Productions

An important feature of this system is that all specifications necessary to define a programming language are in the form of operator definitions. A single operator definition is expressed in a meta-language statement called a *production*; productions are the only statements in the meta-language.

Definition: A production is a cluster of information which defines an operator and associates it with a token of the defined language. A production defining the operator *op* for the token OP must be in one of four forms depending on the operator type.

OPERATOR TYPE	PRODUCTION
NILFIX	"OP" <*p*> ,<rbp>;<denotation>
PREFIX	"OP" ~ <*p*> ,<rbp>;<denotation>
POSTFIX	~ "OP" <*p*> ,<lbp>,<rbp>;<denotation>
INFIX	~ "OP" ~ <*p*> ,<lbp>,<rbp>;<denotation>

where:

1) quotes (") are meta-language symbols enclosing the token being defined.

2) ~ is a meta-language symbol denoting the presence of an argument.

3) <*p*> is an optional annotation pattern, defined in the next paragraph.

4) <lbp> and <rbp> are left and right binding powers, non-negative integers.

5) <denotation> is a program which calculates the translation of *op*, given the translations of its arguments.

<u>**Notation**</u>: When an operator *op* has been defined we refer to the components of the production as follows:

type [*op*] is one of {NILFIX,PREFIX,POSTFIX,INFIX}.

p [*op*] is the annotation pattern defined for *op*.

lbp [*op*] is the left binding power defined for *op*, if any.

rbp [*op*] is the right binding power defined for *op*.

den [*op*] is the denotation defined for *op*.

Aside from patterns, what we have is a simple formalism in which ordinary unary and binary arithmetic operators may be defined. The first part of each production is a template in which the defined operator is quoted and the symbol ~ is a place holder for arguments. The left and right binding powers are stated separately, and the denotation incorporates a translation rule. We recall the production in Example 1 of Section III.A in which the operator + is defined:

Ex. 1 `~ "+" ~ ,14,14;<denotation>`

In this case type [+] = INFIX, and lbp [+] = rbp [+] = 14.

The optional use of annotation patterns is a distinguishing feature of this meta-language. A pattern allows an operator to take multiple right arguments, each labelled with an identifying token. In addition, tokens may be included which label no argument but play a purely syntactic role.

Definition: An annotation pattern, or simply pattern, is an expression specifying possible labelled argument configurations. We use the meta-variables p, q, and r to represent patterns. A pattern p must be in one of the following forms:

1. λ

2. "d" where d is a token

3. "d" ~ where d is a token, ~ a meta-symbol as above

or, inductively, for some annotation patterns q and r, and the associated sets $first_q$, $first_r$, and $cont_q$:

4. qr if $cont_q \cap first_r = \phi$

5. $(q|r)$ if $first_q \cap first_r = \phi$

6. $(q)*$ if $cont_q \cap first_q = \phi$

Definition: A delimiter is a token used in a pattern. We use the meta-variable d (possibly subscripted) to represent a delimiter.

Before defining the sets *first* and *cont*, we refer briefly to Example 2 of Section III.A:

Ex. 2 "IF" ~ "THEN" ~ ("ELSE" ~ | λ) ,6;<denotation>

In this example the operator IF is defined with type[op] = PREFIX, and we have the pattern p[IF] = "THEN" ~ ("ELSE" ~ | λ) (which will be seen to satisfy the restrictions). As in the operator part of a production quotes enclose the tokens, in this case delimiters, of the defined language, and ~ holds the place of an argument.

With the exception of the restrictions imposed on cases 4, 5, and 6, these patterns are ordinary regular expressions with the usual interpretation; the symbols λ, |, and * denote the empty string, pattern alternation, and pattern star closure respectively.

Although the symbol ~ is intended to hold the place of an argument (a subexpression) we will expedite our discussion of patterns by considering a language in which we include the symbol ~ to match itself. Thus, we will say that the string d matches the pattern "d", and the string d ~ matches the pattern "d" ~. Two strings which match p[IF] are THEN ~ and THEN ~ ELSE ~.

Notation: When the string ω matches the pattern p, we write $\omega \propto p$.

Recalling the restrictions imposed in our definition of annotation patterns, we now define the sets *first* and *cont*. We begin by defining our notion of *first*.

Definition: $first(\omega)$ = the first symbol of the string ω (undefined if $\omega = \lambda$).

Definition: $first_p = \bigcup_{\omega \propto p,\, \omega \neq \lambda} \{first(\omega)\}$.

The set $first_p$ is simply the generalization of $first(\omega)$ to all strings matching p. Similarly, we have two forms of *cont*.

Definition: If $\omega \propto p$ then $cont_p(\omega) = \bigcup_{\omega\beta \propto p,\, \beta \neq \lambda} \{first(\beta)\}$.

Definition: $cont_p = \bigcup_{\omega \propto p} cont_p(\omega)$.

The set $cont_p(\omega)$ includes any symbol which may follow ω in a longer string, when *both* ω and the longer string match p. In the context of finding a string to match a particular pattern, this set has the following interpretation. Assume you are scanning a string from left to right and have just reached the end of a string ω which matches the pattern p. If any of the symbols of $cont_p(\omega)$ occur next in the string, then it may be possible to continue scanning and find an extension of ω which also matches p. Referring again to Example 2, we have $cont_{p\text{[IF]}}$(THEN ~) = {ELSE} and $cont_{p\text{[IF]}}$(THEN ~ ELSE ~) = ϕ. The set $cont_p$ is the generalization to include all tokens which might occur this way, so we have $cont_{p\text{[IF]}}$ = {ELSE}.

The sets *first* and *cont* enable us to state important restrictions on the use of patterns, restrictions which are directly motivated by our parsing algorithm for matching strings to patterns. While we can not in general prevent non-local interaction of annotation patterns (e.g., nested IF-THEN-ELSE expressions), it is possible to insure that there are no ambiguities or unexpected results relative to a single pattern. The three restrictions prevent any such problems.

The essence of the matching algorithm is as follows: look at the part of the pattern remaining to be matched and decide what to do next. If the pattern is λ, then simply stop. If it is "d", then look at the next symbol in the input string. It must be d or there is an error. Likewise, "d" $\sim$ means to check for d and afterwards "collect an argument". When the pattern is the alternation $(q \mid r)$, there is the obvious problem of choosing which pattern to use. The decision is made by examining the next symbol and determining whether it is in the sets $first_q$ or $first_r$. When the pattern is qr, the two patterns are simply matched in order. Finally, when the pattern is $(q*)$, the next symbol is always checked for membership in $first_q$. If true, the pattern q is matched and the process repeated.

The restrictions on patterns insure that the choices made by this method are always unique; i.e. that they are the only possible choices. Thus, in the case of $(q \mid r)$ we require that $first_q \cap first_r = \phi$, no symbol may be in both sets. The problem with qr is slightly more subtle; the restriction here ($cont_q \cap cont_r = \phi$) insures that the choice, whether to continue matching a longer string to q, or to stop and begin matching r, is always unique. The * operator is essentially an extension of concatenation, so the restriction on the pattern $(q)*$ is similar. It must always be clear whether to continue matching an instance of q, or to go on to the next, so we require that $cont_q \cap first_q = \phi$. Important properties of these restrictions, independent of any parsing algorithm, are proven in Section V.B.

Sets of Productions

We have now defined the local properties of a meta-language production; there is no other form of definitional information. A complete language definition is any set of productions, defining a set of operators, which satisfies minor global restrictions (to insure that all properties are well-defined).

Definition: An operator is of type NUL-TYPE if it is defined without a left argument.

An operator is of type LEF-TYPE if it is defined with a left argument.

Definition: A language definition D is a set of productions in which:

1) no token OP has more that one NUL-TYPE production,

2) no token OP has more than one LEF-TYPE production, and

3) no token is both a LEF-TYPE operator and a delimiter.

Conditions 1 and 2 allow a token to represent two operators in the special case where one operator takes a left argument and the other does not; i.e., when there will be no ambiguity. In this case, two separate operations are actually being defined, but they are represented by the same symbol. Such an token is both LEF-TYPE and NUL-TYPE. Context, i.e. the presence of the the left argument, will always make it clear which operator is meant. Condition 3 guarantees that the left binding power of every token is well defined, since the parser uses the convention that delimiters have lbp = 0. The left binding power of all delimiters is by convention 0.

IV.B Generative Model

We have presented in Section IV.A the structure of our meta-language. A generative model is now defined which determines the correspondence between a language definition D (a set of productions) and the languages defined by D (a set of token strings). The model is closely related to the assumptions on which the CGOL approach to translator writing is based: the argument relationship among operators and the syntactic conventions for linear representation are related but separate issues.

We begin with the set E_D of abstract expressions, collections of operators with specified argument relationships. We then define three properties of expressions which measure potential for syntactic interaction. Given these properties, we define the subset $E'_D \subseteq E_D$ of expressions which are grammatical; i.e., may be unambiguously represented as linear strings of tokens. The process of linear representation is defined as the function W_D, mapping expressions into the set Σ^* of strings of tokens.

Expression Trees

Our basic notion of abstract expression is based on the relationship "is an argument of" among operators. This notion is extended by ordering and labelling each instance of the relationship, identifying the particular role being played by the argument. Thus, an instance of the relationship might be "is a left argument of" or "is an ELSE argument of".

Our formal model of these expressions is a set of ordered trees with labels on both nodes and branches. A node corresponds to an operator whose arguments (subtrees attached by ordered, labelled branches) occur in a configuration appropriate to the definition of the operator. Examples of these trees are given in Figure 1. Figure 1a is an expression tree containing only arithmetic operators. Arguments here are labelled "left" and "right", indicating their roles. Figure 1b shows a conditional expression in which the test is the "right" argument and the alternative values are appropriately labelled. Figures 1c and 1d illustrate possible uses of delimiters which label no arguments. In these cases the tokens) and FI are included to signal the end of the expression. We formalize this latter technique by permitting labelled branches which connect to the null subtree, although we will not include the null tree as part of our set.

We now define formally the set of expressions corresponding to a meta-language definition. Our basic requirement is that the argument configuration for each operator be appropriate to its definition. This requires a more precise definition of the correspondence between patterns and sequences of subtrees.

<u>**Definition**</u>: The ordered subtrees $e_1, \ldots, e_n$ $(n \geq 0)$, labelled $d_1, \ldots, d_n$, <u>match</u> the pattern p iff one of the following is true:

1. $p = \lambda$ and $n=0$, i.e. there are no subtrees.

2. $p =$ "d" and $n=1$, where e_1 is null and $d=d_1$.

3. $p =$ "d" $\sim$ and $n=1$, where e_1 is non-null and $d=d_1$.

or, where q and r are patterns, one of the following:

4. $p = qr$ and $\exists k$ $0 \leq k \leq n$ such that $e_1, \ldots, e_k$ match q, and

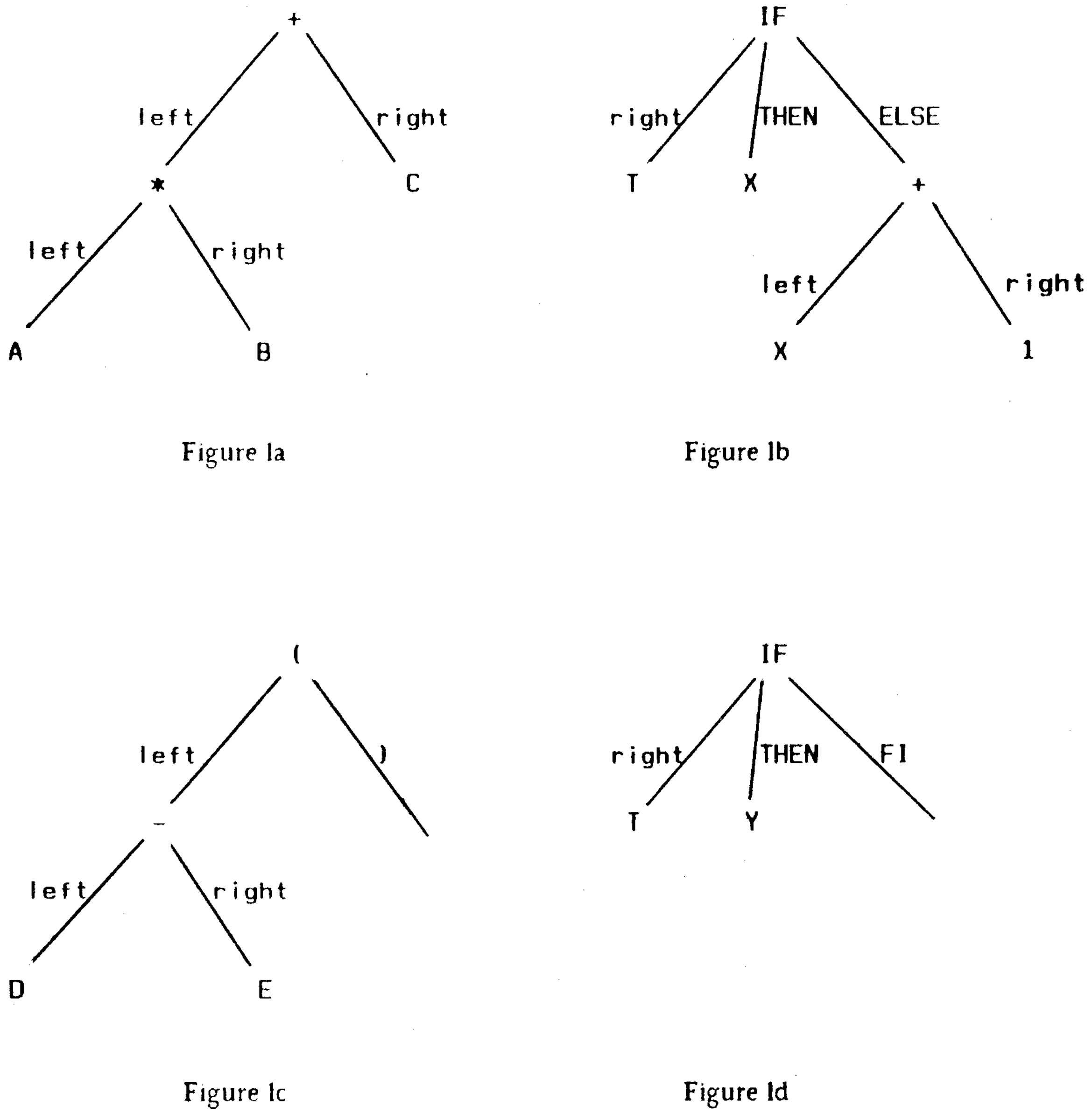

Figure la

Figure lb

Figure lc

Figure ld

Figure l. Expression trees

$e_{k+1}, \ldots, e_n$ match r.

5. $p = (q \mid r)$ and $e_1, \ldots, e_n$ match q or r.

6. $p = (q)*$ and either $n=0$ or $\exists k$ $0<k\leq n$ such that $e_1, \ldots, e_k$ match q, and $e_{k+1}, \ldots, e_n$ match $(q)*$.

We are now ready to define our complete set of expression trees.

Definition: The set of expression trees E_D corresponding to a language definition D contains the set of finite trees defined inductively by:

Basis: $e \in E_D$, where e is a single node with no branches attached, iff the node has a label op such that op is defined in D, type$[op]$=NILFIX, and $\lambda \prec p[op]$.

Induction: $e \in E_D$, where e is a tree with subtrees attached by labelled branches, iff the root node has a label op such that op is defined in D, each non-null subtree is in E_D, and one of the following cases holds:

1. type$[op]$=NILFIX and e has subtrees $e_1, \ldots, e_n$ $(n\geq 0)$, labelled $d_1, \ldots, d_n$, which match $p[op]$.

2. type$[op]$=PREFIX and e has subtrees $e_0, e_1, \ldots, e_n$ $(n\geq 0)$, labelled right, $d_1, \ldots, d_n$, where $e_1, \ldots, e_n$ match $p[op]$.

3. type$[op]$=POSTFIX and e has subtrees $e_{left}, e_1, \ldots, e_n$ $(n\geq 0)$, labelled left, $d_1, \ldots, d_n$, where $e_1, \ldots, e_n$ match $p[op]$.

4. type$[op]$=INFIX and e has subtrees $e_{left}, e_0, e_1, \ldots, e_n$ $(n\geq 0)$, labelled left, right, $d_1, \ldots, d_n$, where $e_1, \ldots, e_n$ match $p[op]$.

Syntactic Properties of Expressions

Having defined our abstract domain of expressions, we now apply syntactic conventions. We claimed in the previous chapter that there are only two basic types of syntactic interaction possible among expressions in linear form. We define three properties of expressions (r-index, l-index, and c-set) which explicitly measure the tendency for an expression to participate in such interactions.

The first and most common form of syntactic interaction is the association of intervening subexpressions (arguments). For example, in the expression a+b*c there is a choice, governed by convention, for the association of the subexpression b; it may either associate to the left (and become an argument to +) or to the right (as an argument to *). Since operators are subject to this interaction on either side (and binding powers may differ from right to left), we define two corresponding properties, beginning with the left.

Definition: If $e \in E_D$ then l-index(e) is defined inductively as follows:

Basis: If e is a node with no branches attached, then

$$\text{l-index}(e) = \infty.$$

Induction: If e has subtrees then let *op* be the label of the root node:

a) if *op* is of type LEF-TYPE (i.e. if it has a left argument), then

$$\text{l-index}(e) = \min[\text{lbp}[op], \text{l-index}(e_{left})],$$

b) otherwise (if no left argument)

$$\text{l-index}(e) = \infty.$$

The value of l-index is a numerical measurement of an expression's resistance to false association to the left. If an operator has no left argument at all, then there can never be an intervening expression on the left, so there can never be a problem. In this case, l-index is ∞. Since a left argument may itself be an expression with a left argument, and so on, this property is defined inductively over all such subexpressions. An expression's resistance is only as high as the weakest "exposed" operator.

For example, if e is the expression tree shown in Figure 1a, there are two operators exposed to the left, + and *. By definition we know that

$$\text{l-index}[e] = \min[\text{lbp}[+], \text{lbp}[*], \text{lbp}[A]],$$

but since $\text{lbp}[A] = \infty$ this is equal to $\min[\text{lbp}[+], \text{lbp}[*]]$. From this we understand that we have two subexpressions, A and A*B, which might be falsely associated to the left. In an expression tree these exposed operators are those which may be reached from the top by following branches labelled left down the tree.

The situation on the right side of expressions is analogous although complicated slightly by the presence of multiple right arguments.

<u>Definition</u>: If $e \in E_D$ then <u>r-index(e)</u> is defined inductively as follows:

Basis: If e is a node with no branches attached, then

$$\text{r-index}(e) = \infty.$$

Induction: If e has subtrees, then let *op* be the label of the root node:

a) if there is a subtree e_n, and if it is non-null, then

$$\text{r-index}(e) = \min[\text{rbp}[op], \text{r-index}(e_n)],$$

b) otherwise

$$\text{r-index}(e) = \infty.$$

The value of r-index is analogous to l-index except that we now refer to e_n instead of e_{left}. When there are subtrees $e_1, \ldots, e_n$ which match the pattern $p[op]$ (i.e., when $n>0$) then e_n is simply the last (rightmost) one. It is this subexpression which, if non-null, is exposed to the right and is subject to false association. For example, both 1 and X+1 are exposed to the right in the expression of Figure Ib. If e_n is null, then its label d_n is being used as a purely syntactic token to indicate that there are no more arguments to the right. In this case there is no possibility of false association, so the value of r-index is ∞. Examples of this are the expressions of Figures Ic and Id. Now if the annotation part of the expression is entirely null (i.e. $n=0$), then the expression is of the ordinary arithmetic variety (e.g., Figure Ia). In this case, e_n refers to the right argument e_0, if there is one, and r-index is the exact counterpart of l-index.

We turn now to the second type of syntactic interaction, the generalized dangling ELSE problem. We recall that our pattern matching algorithm (i.e. for collecting right arguments) will continue to gather arguments as long as possible. We are interested in the case where the pattern p has been matched (say by the string ω) and there is a choice whether to continue. Any token for which this is possible is by definition in $cont_p(\omega)$. Looking at our standard example where p[IF] = "THEN" ~ ("ELSE" ~ | λ), we have:

$$cont_{p[\mathrm{IF}]}(\text{ THEN } \sim\ ) = \{\mathrm{ELSE}\}.$$

This tells us that if the operator IF has so far collected the token THEN and a following argument then the collection may stop, but if ELSE appears next in the input string, it will be included. When we deal with general expression trees, this problem can be caused either at the top level (by the pattern of the topmost operator) or at lower levels (in exposed rightmost arguments), so the property c-set is defined recursively, similar to r-index. The c-set of an expression is the set of all delimiters which would be incorrectly included if placed after the expression in linear form.

This definition requires the property *cont* to be defined on an ordered set of subtrees, rather than the on strings of the original definition. The correspondence is quite straightforward: a null subtree e_i with branch labelled d_i corresponds to the single symbol d_i, and a non-null subtree e_j labelled d_j corresponds to the string d_j ~. As is proven in Lemma 11 of Section V.B, this translation does not affect the definition of *cont*; the symbol ~ can never be in the set.

<u>Definition</u>: If $e \in E_D$ then <u>c-set(e)</u> is defined inductively as follows:

Basis: If e is a node with no branches attached, then

$$\text{c-set}(e) = cont_{p[op]}(\lambda).$$

Induction: If e has subtrees, then let op be the label of the root node:

a) if there is a subtree e_n and if it is non-null, then

$$\text{c-set}(e) = cont_{p[op]}(e_1, \ldots, e_n) \cup \text{c-set}(e_n),$$

b) otherwise

$$\text{c-set}(e) = cont_{p[op]}(e_1, \ldots, e_n).$$

<u>Grammatical Expressions</u>

We now use these three syntactic properties to restrict our set E_D of expressions by eliminating those which permit unwanted syntactic interactions.

<u>Definition</u>: $e \in E_D$ is <u>grammatical</u> iff one of the following is true:

Basis: e has no branches attached, or

Induction: e is a tree with root node labelled op and with subtrees satisfying:

0) each non-null subtree is grammatical,

1) $\text{r-index}(e_{left}) \geq \text{lbp}[op]$, if there is a subtree e_{left}.

2) $\text{rbp}[op] < \text{l-index}(e_i)$, for $0 \leq i \leq n$, when e_i is non-null.

3) $d_i \in \text{c-set}(e_{i-1})$, for $1 \leq i \leq n$, when e_i is non-null.

This definition allows us to build trees while watching for syntactic problems. The restrictions correspond to the informal rules described in Section III.B; each restriction may be understood as the prevention of unwanted syntactic interaction. Restriction 1 covers the use of an expression as a left argument; it insures that the whole expression will be treated as the argument, not some exposed fragment. For example, this restriction would prevent the use of the expression in Fig. 1a as a left argument to the operator ↑, since the subexpression C would incorrectly become the left argument of ↑. Restriction 2 is the equivalent on the right side. Restriction 3 insures that no delimiter will be improperly included with a subexpression; e.g., don't use an IF-THEN expression as the THEN argument to an IF-THEN-ELSE expression.

Definition: $E'_D = \{e \in E_D \mid e \text{ is grammatical}\}$.

Our defined language will be based on only the expression trees which are grammatical. Ungrammatical trees may be easily fixed by the addition of some operator with bracketing properties, typically the semantically null operator (. For example, the expression shown in Figure 1c would, given a reasonable definition, have **lbp = rbp = 0** and *cont* = ϕ; i.e., it is syntactically secure.

The Writing Function

Now that we have eliminated syntactic problems from our set of expressions, we may use a trivial writing function.

Definition: The writing function W_D is defined recursively on the set E_D as follows:

If $e \in E_D$ then: $W_D(e) = \alpha \text{ OP } \beta \; d_1\gamma_1 \ldots d_n\gamma_n$ where:

OP is the token naming the operator at the root node of e.

$\alpha = W_D(e_{left})$ if e_{left} exists, λ otherwise.

$\beta = W_D(e_0)$ if e_0 exists, λ otherwise.

d_i = the label on tree e_i for $1 \le i \le n$.

$\gamma_i = W_D(e_i)$ for $1 \le i \le n$, when e_i is non-null, λ otherwise.

The linear representation of trees defined by W_D uses a very simple convention. An argument is preceded by its label, with two important exceptions: the labels **left** and **right** are implicitly represented by juxtaposition with the operator.

The Defined Language

Finally, the defined language S_D is simply the linear form of the grammatical trees.

Definition: Given a set D of productions, the defined language is S_D, where

$$S_D = W_D(E'_D).$$

IV.C The Parsing Program

We present the parsing program in two parts; in addition to the actual program (which we will view as a function from strings into expression trees) we give a specification of the internal representation required for meta-language productions. A program which automatically converts a meta-language production to this internal form is called the *meta-language processor.*

The Meta-language Processor

There is virtually no processing of the information given in the productions of the meta-language. It is simply broken into the natural categories, converted into a standard LISP representation, and attached to the property list of the defined token. The categories and their property list names are:

1. Type (e.g. INFIX)	NUL-TYP, LEF-TYP
2. Annotation Pattern	NUL-PAT, LEF-PAT
3. Left binding power	LBP
4. Right binding power	NUL-RBP, LEF-RBP
5. Denotation	NUL-DEN, LEF-DEN

Since it is possible to have two operators for the same token, one with a left argument and one without, the two sets of data will be separately named so they may coexist and be independently retrieved from the property lists. The one exception is the left binding power, since it is irrelevant for NUL-TYPE operators. Any token used as a delimiter, however, will have its left binding power set to 0. The denotation properties will not be used in this implementation, since it will only parse and not translate.

The definitional information will be represented as LISP data in the following forms:

Type: The NUL-TYP and LEF-TYP properties are simply the appropriate names. Thus NUL-TYP may be either NILFIX or PREFIX, and LEF-TYP may be either POSTFIX or INFIX.

Left binding power: The property is a non-negative integer.

Right binding power: The property is a non-negative integer.

Annotation Pattern: The representation of a pattern p is the list repr $[p]$ defined recursively by:

1. If $p = \lambda$ then repr $[p]$ = (LAMB).

2. If p = "d" then repr $[p]$ = (d) , where d is the token.

3. If p = "d" ~ then repr $[p]$ = (d ARG) , where d is the token.

or, if q and r are patterns, and repr $[q]$ and repr $[r]$ their representations:

4. If $p = q r$ then repr $[p]$ = (CONC repr $[q]$ repr $[r]$).

5. If $p = (q | r)$ then repr $[p]$ = (UNION repr $[q]$ repr $[r]$).

6. If $p = (q)*$ then repr $[p]$ = (STAR repr $[q]$).

Since this information is on property lists, it is globally available to the parsing program; a request for one of these properties will have the same value independent of the particular environment from which it is made. For the purposes of proof, we give the following axioms which formally specify the operation of the meta-language processor.

Axiom 1: If the token OP is defined in D as a nilfix operator, then

(a) (GET 'OP 'NUL-TYP) = NILFIX

(b) (GET 'OP 'NUL-PAT) = repr $[p[op]]$

(c) (GET 'OP 'NUL-RBP) = rbp $[op]$

__Axiom 2__: If the token OP is defined in D as a prefix operator, then

(a) (GET 'OP 'NUL-TYP) = PREFIX

(b) (GET 'OP 'NUL-PAT) = repr[*p*[*op*]]

(c) (GET 'OP 'NUL-RBP) = rbp[*op*]

__Axiom 3__: If the token OP is defined in D as a postfix operator, then

(a) (GET 'OP 'LEF-TYP) = POSTFIX

(b) (GET 'OP 'LEF-PAT) = repr[*p*[*op*]]

(c) (GET 'OP 'LEF-RBP) = rbp[*op*]

__Axiom 4__: If the token OP is defined in D as an infix operator, then

(a) (GET 'OP 'LEF-TYP) = INFIX

(b) (GET 'OP 'LEF-PAT) = repr[*p*[*op*]]

(c) (GET 'OP 'LEF-RBP) = rbp[*op*]

__Axiom 5__: If the token OP is used as a delimiter in any production in D, then

(5) (GET 'OP 'LBP) = 0

It may now be seen how our global restrictions on sets of productions insure that all of these properties are well-defined. Properties NUL-TYP, NUL-PAT, NUL-RBP, and NUL-DEN can only be determined if a nilfix or prefix operator is defined for OP, but we only allow one such production per token. Similarly, LEF-TYP, LEF-PAT, LEF-RBP, and LEF-DEN are well-defined. LBP may only be determined if a postfix or infix operator is defined for OP (in which case only one such definition is allowed) or if it is used anywhere as a delimiter (in which case the LBP is 0, no matter how many times it is used). A token may not, however, be both.

The Parsing Program

We present below the LISP code for a straightforward parser implementation. The parser returns the expression tree in a simple list representation defined below; an extension to the full translator would have the arguments passed to the denotation, rather than being assembled into a list.

Expression Tree: The representation of a tree $e \in E_D$ is the recursively defined list:

repr[e] = (OP r_{left} r_{right} r_1 ... r_n) where

r_{left} = (LEFT repr[e_{left}]) if e_{left} exists, otherwise non-existent

r_{right} = (RIGHT repr[e_θ]) if e_θ exists, otherwise non-existent

r_i = (d_i repr[e_i]) if e_i is non-null

r_j = (d_j) if e_j is null

Several prominent features of this program should be kept in mind; it was written for perspicuity and convenience of proof. There are therefore no global variable references; for each subroutine the input stream is passed as an argument and returned as a value. The result is a program which is approximately twice as long and much less efficient than it could be. The main problem is that passing the input string as an argument often requires that the same expression be evaluated more than once. This problem could be easily solved but would result in rather more obscure code; efficiency has been sacrificed for clarity. An equivalent but efficient program could be proven correct by proving its equivalence to this one. Such a proof should be considerably shorter than an original proof of correctness as given here.

The Basic Parsing Program

```
(DEFUN PARSE (RBP STRING)
        (ASSOC RBP (NUL-TYPE STRING)))


(DEFUN ASSOC (RBP STATE)
        (COND ((LESSP RBP (GET (CADR STATE) 'LBP))
               (ASSOC RBP (LEF-TYPE STATE)))
              (T STATE)))
```

This is the top level control structure of the parser. The function PARSE receives as input a right binding power and a list of symbols, the string in S_D to be parsed. The status of the parse is contained in the variable STATE which is passed and returned among the procedures. STATE is always a list whose first element is the representation of the expression (tree) parsed so far, and whose remaining elements are the unparsed input string. Given that an expression has been parsed, the function ASSOC (not the standard LISP function ASSOC) decides whether to give it as a left argument to the next operator in the string (by calling ASSOC recursively), or to return the current state.

The function NUL-TYPE collects the arguments for the next operator in the string, on the assumption that it is nilfix or prefix. It in turn calls NILFIX or PREFIX to handle the separate cases. The function LEF-TYPE is similar, except that the expression parsed so far is assumed to be the left argument to the next operator in the string. The subroutine FIND handles the collection of all annotation tokens and arguments; it uses the functions LAMBDA-P (predicate for null string membership in a pattern) and FIRST (the set *first* previously defined).

Functions to Process NUL-TYPE Operators

```
(DEFUN NUL-TYPE (STRING)
       (COND ((NULL (CDDR STATE)) ERROR)               ;end of input
             ((EQ (GET (CAR STRING) 'NUL-TYP) 'NILFIX)
              (NILFIX (CAR STRING)                     ;operator
                      (CDR STRING)                     ;unparsed string
                      (GET (CAR STRING) 'NUL-RBP)      ;rbp[op]
                      (GET (CAR STRING) 'NUL-PAT)))    ;p[op]
             ((EQ (GET (CAR STRING) 'NUL-TYP) 'PREFIX)
              (PREFIX (CAR STRING)                     ;as above
                      (CDR STRING)
                      (GET (CAR STRING) 'NUL-RBP)
                      (GET (CAR STRING) 'NUL-PAT)))
             (T                                        ;default case
              (NILFIX (CAR STRING)                     ;variable or
                      (CDR STRING)                     ;constant
                      0
                      '(LAMB))) ))


(DEFUN NILFIX (OPERATOR REST RBP PAT)
       (CONS (APPEND (LIST OPERATOR)
                     (CAR (FIND RBP (CONS NIL REST) PAT)))
             (CDR (FIND RBP (CONS NIL REST) PAT)) ))


(DEFUN PREFIX (OPERATOR REST RBP PAT)
  (CONS (APPEND (LIST OPERATOR)
                (LIST (LIST 'RIGHT (CAR (PARSE RBP REST))))
                (CAR (FIND RBP
                           (CONS NIL (CDR (PARSE RBP REST)))
                           PAT)))
        (CDR (FIND RBP (CONS NIL (CDR(PARSE RBP REST)))PAT))))
```

Functions to Process LEF-TYPE Operators

```
(DEFUN LEF-TYPE (STATE)
       (COND ((NULL (CDDR STATE)) ERROR)                ;end of string
             ((EQ (GET (CADR STATE) 'LEF-TYP) 'POSTFIX)
              (POSTFIX (CAR STATE)                       ;left arg
                       (CADR STATE)                      ;operator
                       (CDDR STATE)                      ;unparsed string
                       (GET (CADR STATE) 'LEF-RBP)       ;rbp[op]
                       (GET (CADR STATE) 'LEF-PAT) ))    ;p[op]
             ((EQ (GET (CADR STATE) 'LEF-TYP) 'INFIX)
              (INFIX (CAR STATE)                         ;as above
                     (CADR STATE)
                     (CDDR STATE)
                     (GET (CADR STATE) 'LEF-RBP)
                     (GET (CADR STATE) 'LEF-PAT) ))
             (T ERROR) ))                                ;no left def.


(DEFUN POSTFIX (LVAL OPERATOR REST RBP PAT)
       (CONS (APPEND (LIST OPERATOR)
                     (LIST (LIST 'LEFT LVAL))
                     (CAR (FIND RBP (CONS NIL REST) PAT)))
             (CDR (FIND RBP (CONS NIL REST) PAT))))


(DEFUN INFIX (LVAL OPERATOR REST RBP PAT)
  (CONS (APPEND (LIST OPERATOR)
                (LIST (LIST 'LEFT LVAL))
                (LIST (LIST 'RIGHT (CAR (PARSE RBP REST))))
                (CAR (FIND RBP
                           (CONS NIL (CDR (PARSE RBP REST)))
                          PAT)))
        (CDR (FIND RBP (CONS NIL (CDR(PARSE RBP REST)))PAT))))
```

Annotation Argument Processor

```
(DEFUN FIND (RBP STATE PAT)
  (COND ((EQ (CAR PAT) 'LAMB)                              ;p=λ
         STATE)
                                                            ;p=qr
        ((EQ (CAR PAT) 'CONC)
         (FIND RBP (FIND RBP STATE (CADR PAT)) (CADDR PAT)))
                                                            ;p=(q|r)
        ((EQ (CAR PAT) 'UNION)
         (COND ((MEMBER (CADR STATE) (FIRST (CADR PAT)))
                (FIND RBP STATE (CADR PAT)))
               ((MEMBER (CADR STATE) (FIRST (CADDR PAT)))
                (FIND RBP STATE (CADDR PAT)))
               ((LAMBDA-P PAT)
                STATE)
               (T ERROR)))         ;neither alternative matches
                                                            ;p=(q)*
        ((EQ (CAR PAT) 'STAR)
         (COND ((MEMBER (CADR STATE) (FIRST (CADR PAT)))
                (FIND RBP (FIND RBP STATE (CADR PAT)) PAT))
               (T STATE)))
                                                            ;p="d"
        ((AND (NULL (CDR PAT)) (EQ (CAR PAT) (CADR STATE)))
         (CONS (APPEND (CAR STATE)
                       (LIST (LIST (CADR STATE))))
               (CDDR STATE)))
                                                            ;p="d" ~
        ((EQ (CAR PAT) (CADR STATE))
         (CONS (APPEND (CAR STATE)
                       (LIST (LIST (CADR STATE)
                                   (CAR (PARSE RBP (CDDR STATE))))))
               (CDR (PARSE RBP (CDDR STATE)))))
        (T ERROR)))            ;missing token-- (car pattern)
```

Pattern Processing Functions

```
(DEFUN LAMBDA-P
        (P)
        (COND ((EQ (CAR P) 'LAMB) T)                              ;p=λ
               ((EQ (CAR P) 'CONC)                                ;p=qr
                (AND (LAMBDA-P (CADR P)) (LAMBDA-P (CADDR P))))
               ((EQ (CAR P) 'UNION)                               ;p=(q|r)
                (OR (LAMBDA-P (CADR P)) (LAMBDA-P (CADDR P))))
               ((EQ (CAR P) 'STAR) T)                             ;p=(q)*
               (T NIL)))                                          ;p="d" or "d"~


(DEFUN FIRST
        (P)
        (COND ((EQ (CAR P) 'LAMB) NIL)                            ;p=λ
               ((EQ (CAR P) 'CONC)                                ;p=qr
                (APPEND (FIRST (CADR P))
                        (COND ((LAMBDA-P (CADR P))  (FIRST (CADDR P)))
                              (T NIL))))
               ((EQ (CAR P) 'UNION)                               ;p=(q|r)
                (APPEND (FIRST (CADR P)) (FIRST (CADDR P))))
               ((EQ (CAR P) 'STAR) (FIRST (CADR P)))              ;p=(q)*
               (T (LIST (CAR P)))))                               ;p="d" or "d"~
```

V. CORRECTNESS

Using the definitions presented in Chapter IV, we are now prepared to formally state and prove the notion of correctness discussed informally in Section III.D. In the first section of this chapter we state our main result, the PARSE theorem, and discuss three important corollaries which embody more closely our intuitive notions of correctness. Section V.B presents a number of preliminary lemmas, dealing primarily with properties of annotation patterns in our meta-language. These results are theoretical properties and are completely independent of the parsing algorithm. Sections V.C and V.D contain the proofs of parts I and II, respectively, of the PARSE theorem; these theorems are long but straightforward since the interesting theoretical results are separately proven.

V.A Formal Statement

We begin a formal statement of correctness by recalling the user-oriented description of a defined language. For any set of meta-language productions D, the language $S_D \subseteq \Sigma^*$ defined by D is $S_D = W_D(E'_D)$, where W_D is the writing function and E'_D is the set of grammatical expression trees. The parser for the language, constructed by the algorithm of Section IV.C, is represented by the function P_D. This function maps strings of Σ^* into expression trees (defined in IV.C). The function P_D is partial; when we write $P_D(\delta) = e$, we mean that the parser, when given the input string δ, halts error-free and returns e. We state now our main result.

<u>**PARSE THEOREM**</u>:

I. $\forall D \ \forall e \in E'_D \ (P_D(W_D(e)) = e)$

II. $\forall D \ \forall \delta \in \Sigma^* \ (P_D(\delta)$ halts error-free $\Rightarrow \delta \in S_D)$

For the rest of this chapter we assume that D refers to some language definition expressed in the meta-language of Section IV.A; i.e., we drop the "for all D".

We examine now the sufficiency of the result relative to general notions of correctness in the form of corollaries. The first is that a translator should be an *acceptor* for the language S_D in the ordinary sense: the translator should halt error-free *exactly* when given a sentence in the language S_D.

Corollary 1 (Acceptor):

$$\forall s \in \Sigma^* \ (P_D(s) \text{ halts error-free} \Leftrightarrow s \in S_D)$$

Proof: One direction simply restates part II of the PARSE theorem. Now assume $s \in S_D$. By definition there is some $e \in E'_D$ such that $s = W_D(e)$. Part I says $P_D(W_D(e)) = P_D(s) = e$; i.e., PARSE halts error-free.∎

We also expect that the translator, when it halts error-free, returns a valid parse of the input string.

Corollary 2 (Parser):

$$\forall s \in S_D \ (P_D(s) \in E'_D \wedge W_D(P_D(s)) = s)$$

Proof: Assume $s \in S_D$. Then there is some $e \in E'_D$ such that $s = W_D(e)$. By Part I we know $P_D(s) = P_D(W_D(e)) = e \in E'_D$. Furthermore, since $P_D(s) = e$, we have $W_D(P_D(s)) = W_D(e) = s$.∎

We note that Corollary 2 only guarantees the output of *some* valid expression tree, or parse, for each input. We have not proven that such a parse must be unique; i.e., that the language is unambiguous. Ambiguity is a property of a language and its means of definition, not of a particular parsing scheme.

Corollary 3 (Uniqueness):

$$\forall e, e' \in E'_D \ (W_D(e) = W_D(e') \Rightarrow e = e')$$

Proof: Assume $W_D(e) = W_D(e')$ for $e, e' \in E'_D$. Since the parser is a function,

$\mathsf{P}_\mathsf{D}(\mathsf{W}_\mathsf{D}(e)) = \mathsf{P}_\mathsf{D}(\mathsf{W}_\mathsf{D}(e')$. Then by Part I we have $e = \mathsf{P}_\mathsf{D}(\mathsf{W}_\mathsf{D}(e))$
$= \mathsf{P}_\mathsf{D}(\mathsf{W}_\mathsf{D}(e')) = e'$.∎

Although not strictly a property of the parser, we treat this property here for completeness and convenience of proof.

V.B Preliminary Lemmas

This section formally states and proves a number of necessary properties of our definitional system. Some are merely restatements of definitions and are included for uniform reference; the majority are derived properties which are essential to the program proof. The final two lemmas are correctness proofs of two simple utility programs, LAMBDA-P and FIRST.

We begin with binding powers.

Lemma 1 (Binding powers):

(a) If the token *op* is defined as an operator in D, then rbp[*op*] $\geq$ 0 and lbp[*op*] $\geq$ 0 if defined.

(b) If the token d is used as a delimiter in D, then lbp[d] = 0.

(c) For any $e \in E_D$, l-index[e] $\geq$ 0 and r-index[e] $\geq$ 0.

Proof: Parts (a) and (b) are immediate from the definitions. Part (c) uses part (a) and follows by trivial induction over the definitions of l-index and r-index.∎

The following lemmas describe properties of annotation patterns. Although patterns ultimately determine sequences of labelled subtrees, these properties will be stated and proven in terms of a simpler but equivalent language. We say that a pattern may be matched by strings of symbols, where the symbols include the special symbol ~ and tokens of the defined language. The same convention was used in the discussion of *first* and *cont* in Section IV.A. The correspondence between the strings used here and the ordered sets of labelled subtrees is straightforward. The symbol ~ can only follow a token in strings which match patterns. A token d followed by ~ in one of these strings corresponds to a non-null subtree labelled d. A token d not followed by ~ corresponds to a null subtree labelled d. Lemmas 4 and 11 guarantee that the symbol ~ is invisible; i.e., it plays no role in any of the results presented here. The results apply equally to sequences of labelled subtrees.

For convenience we restate here an essential feature of the definition of patterns, the restrictions on the inductive use of pattern concatenation, alternation, and star closure.

Restrictions (Definition of patterns): Let p, q, r be patterns.

R1. If $p = qr$ then $cont_q \cap first_r = \phi$.

R2. If $p = (q|r)$ then $first_q \cap first_r = \phi$.

R3. If $p = (q)*$ then $cont_q \cap first_q = \phi$.

Because our parsing algorithm continually requires us to treat λ as a special case, we would like to know some of the null-string properties of patterns.

Lemma 2 (λ predicate): Let p, q, r be patterns.

(a) If $p = qr$ then $\lambda \propto p$ iff $\lambda \propto q \wedge \lambda \propto r$.

(b) If $p = (q|r)$ then $\lambda \propto p$ iff $\lambda \propto q \vee \lambda \propto r$.

(c) If $p = (q)*$ then $\lambda \propto p$.

Proof: Immediate from the definition of match.▌

Lemma 2 is the basis for the algorithm used by LAMBDA-P, which calculates whether or not λ matches a particular pattern.

The next lemma is relevant to the computation of the set *first* for a pattern.

Lemma 3 (*first*): Let p, q, r be patterns.

(a) If $p = qr$ then
1. if $\lambda \not\propto q$ then $first_p = first_q$
2. if $\lambda \propto q$ then $first_p = first_q \cup first_r$.

(b) If $p = (q|r)$ then $first_p = first_q \cup first_r$.

(c) If $p = (q)*$ then $first_p = first_q$.

Proof: Immediate from the definitions of *first* and match.■

The parser look at the first symbol of a string in order to decide how to begin matching the string to a pattern. The next lemma guarantees that the parser never looks at $\sim$ when deciding; i.e., that the first symbol is always some delimiter and not part of a subexpression.

Lemma 4: If p is a pattern, $first_p$ contains only tokens (not $\sim$).

Proof: By induction on the definition of a pattern. If $p = \lambda$, "d", or "d" $\sim$ then $first_p = \phi$, {d}, or {d} respectively. If $p = qr$, $(q|r)$, or $(q)*$, then by Lemma 3 and induction $first_p$ contains only tokens.■

We turn now to properties of the set *cont*. We begin with its value relative to the null string.

Lemma 5: If p is a pattern and $\lambda \prec p$ then $cont_p(\lambda) = first_p$.

Proof: From definitions,

$$cont_p(\lambda) = \bigcup_{\lambda\beta \prec p, \beta \neq \lambda} \{first(\beta)\} = \bigcup_{\beta \prec p, \beta \neq \lambda} \{first(\beta)\} = first_p. ■$$

This result has a strong implication for star closure; restriction R3 prevents the use of star closure on nontrivial patterns matched by the null string.

Lemma 6: If p is a pattern and $cont_p \cap first_p = \phi$ and $\lambda \prec p$ then only λ matches p.

Proof: Assume $\lambda \prec p$. By Lemma 5 we have $first_p = cont_p(\lambda) \subseteq cont_p$. Since we assumed $cont_p \cap first_p = \phi$, it must be that $first_p = \phi$, implying that no string other than λ can match p.■

The next lemma is a preliminary result to be used in the proofs of Lemmas 8 and 10. It deals with the way in which a string can match a concatenated pattern.

Definition: The string ω is a prefix of string ω' iff $\omega' = \omega\alpha$ for some string α; if $\alpha \neq \lambda$ then ω is said to be a nontrivial prefix of ω'.

Lemma 7 (Ambiguity): Let q and r be patterns. If

(1) $cont_q \cap first_r = \phi$,

(2) $\omega = \omega_1\omega_2$ where $\omega_1 \prec q$ and $\lambda \neq \omega_2 \prec r$,

(3) $\omega' = \omega_1'\omega_2'$ where $\omega_1' \prec q$ and $\omega_2' \prec r$, and

(4) ω is a prefix of ω',

then $\omega_1 = \omega_1'$.

Proof: Since ω is a prefix of ω', exactly one of the following cases must hold: (i) ω_1 is a nontrivial prefix of ω_1', (ii) ω_1' is a nontrivial prefix of ω_1, or (iii) $\omega_1 = \omega_1'$. We will show that (i) and (ii) do not hold.

(i) $\omega_1' = \omega_1\alpha$ for some $\alpha \neq \lambda$. By definition, $first(\alpha) \in cont_q(\omega_1) \subseteq cont_q$. Since $\omega_2 \neq \lambda$, we also have $first(\alpha) = first(\omega_2) \in first_r$, violating condition (1).

(ii) $\omega_1 = \omega_1'\alpha$ for some $\alpha \neq \lambda$. It cannot be the case that $\omega_2' = \lambda$ because $\omega = \omega_1\omega_2$ is a prefix of ω'. By symmetry this reduces to case 1.∎

It is a corollary of Lemma 7 that when a string ω matches $p = qr$, it matches in only one way. Applying Lemma 7 inductively, we get the same implication for star closure.

Lemmas 8, 9, and 10 describe the contents of the set *cont* relative to concatenation, alternation, and star closure. Since these are the essential lemmas for the actual program proof, they are stated in terms of specific strings; i.e., they describe $cont_p(\omega)$ rather than $cont_p$. The lemmas are intended to directly imply the correctness of the pattern matching part of the parsing algorithm. For example, Lemma 8 guarantees that concatenated patterns may be dealt with locally, one at a time.

Lemma 8 (*cont*): Let $p = qr$ and $\omega = \omega_1\omega_2$ with $\omega_1 \prec q$ and $\omega_2 \prec r$, then

(a) if $\omega_2 \neq \lambda$ then $cont_p(\omega) = cont_r(\omega_2)$

(b) if $\omega_2 = \lambda$ then $cont_p(\omega) = cont_r(\omega_2) \cup cont_q(\omega_1)$.

Proof: (a)

$\supseteq$ We have $cont_r(\omega_2) = \bigcup_{\omega_2\beta \prec r, \beta \neq \lambda} \{first(\beta)\} \subseteq \bigcup_{\omega_1\omega_2\beta \prec p, \beta \neq \lambda} \{first(\beta)\}$ $= cont_p(\omega)$ by definition and since $\omega_2\beta \prec r$ implies that $\omega_1\omega_2\beta \prec p$.

$\subseteq$ By definition $cont_p(\omega) = \bigcup_{\omega\beta \prec p, \beta \neq \lambda} \{first(\beta)\}$. If $\omega\beta = \omega_1\omega_2\beta \prec p$, then let $\omega\beta = \omega' = \omega_1'\omega_2'$ where $\omega_1' \prec q$ and $\omega_2' \prec r$. Since ω is a prefix of ω' and $\omega_2 \neq \lambda$, we have $\omega_1 = \omega_1'$ by Lemma 7. Then $\omega_2' = \omega_2\beta$, so $first(\beta) \in cont_r(\omega_2)$.

(b)

$\supseteq$ As in part (a) $cont_r(\omega_2) \subseteq cont_p(\omega)$. In addition, $cont_q(\omega_1) = \bigcup_{\omega_1\beta \prec q, \beta \neq \lambda} \{first(\beta)\} \subseteq \bigcup_{\omega_1\beta \prec p, \beta \neq \lambda} \{first(\beta)\} = cont_p(\omega)$ since $\omega_1\beta \prec q$ implies that $\omega_1\beta = \omega_1\lambda\beta \prec p$.

$\subseteq$ By definition $cont_p(\omega) = \bigcup_{\omega\beta \prec p, \beta \neq \lambda} \{first(\beta)\}$. If $\omega\beta = \omega_1\omega_2\beta = \omega_1\beta \prec p$ then let $\omega_1\beta = \omega' = \omega_1'\omega_2'$ where $\omega_1' \prec q$ and $\omega_2' \prec r$. We consider the three cases of the relationship between ω_1 and ω_2'.

(i) If ω_1 is a nontrivial prefix of ω_1', then $first(\beta) \in first_r$.

(ii) If ω_1' is a nontrivial prefix of ω_1, then $\omega_2' \neq \lambda$. But then $first(\omega_2') \in cont_q(\omega_1')$. Since $first(\omega_2') \in first_r$, this violates R1.

(iii) If $\omega_1 = \omega_1'$, then $\beta = \omega_2'$ and $first(\beta) \in first_r$. By Lemma 5 $first_r = cont_r(\lambda) = cont_r(\omega_2)$. ▮

Lemma 9 (*cont*): Let p, q, r be patterns and $p = (q \mid r)$.

(a) If $\omega = \lambda$ then $cont_p(\omega) = first_q \cup first_r$.

(b) If $\lambda \neq \omega \prec q$ then $cont_p(\omega) = cont_q(\omega)$.

(c) If $\lambda \neq \omega \prec r$ then $cont_p(\omega) = cont_r(\omega)$.

Proof: (a) By Lemma 5 $cont_p(\lambda) = first_p$, and by Lemma 3b $first_p = first_q \cup first_r$.

(b) Claim $\omega\beta \prec p$ iff $\omega\beta \prec q$. Clearly $\omega\beta \prec q$ implies $\omega\beta \prec p$. Conversely, if $\omega\beta \prec p$ then either $\omega\beta \prec q$ or $\omega\beta \prec r$. If $\omega\beta \prec r$ then $\mathit{first}(\omega\beta) \in \mathit{first}_r$. But since $\omega \neq \lambda$, $\mathit{first}(\omega\beta) = \mathit{first}(\omega) \in \mathit{first}_q$, violating R2, so $\omega\beta \prec q$. We conclude that $\mathit{cont}_p(\omega) = \bigcup_{\omega\beta \prec p, \beta\neq\lambda} \{\mathit{first}(\beta)\} = \bigcup_{\omega\beta \prec q, \beta\neq\lambda} \{\mathit{first}(\beta)\} = \mathit{cont}_q(\omega)$.

(c) Similar to part (b).▮

<u>Lemma 10</u> (*cont*): Let p, q, r be patterns and $p = (q)^*$.

(a) If $\omega=\lambda$ then $\mathit{cont}_p(\omega) = \mathit{first}_q$.

(b) If $\lambda\neq\omega\prec p$ where $\omega=\omega_1\ldots\omega_n$ for $n\geq 1$ and $\omega_i \prec q$ for $1\leq i\leq n$, then

$$\mathit{cont}_p(\omega) = \mathit{cont}_q(\omega_n) \cup \mathit{first}_q$$

<u>Proof</u>:

(a) If $\omega=\lambda$ then $\mathit{cont}_p(\lambda) = \mathit{first}_p = \mathit{first}_q$ by Lemmas 5 and 3c.

(b) By Lemma 6 we need only consider 2 cases: either q is matched by only λ, or q is not matched by λ at all. Since $\omega\neq\lambda$ we assume the second case, where $\lambda \not\prec q$.

$\supseteq$ We have $\mathit{first}_q = \bigcup_{\beta\prec q, \beta\neq\lambda} \{\mathit{first}(\beta)\} \subseteq \bigcup_{\omega\beta\prec p, \beta\neq\lambda} \{\mathit{first}(\beta)\} = \mathit{cont}_p(\omega)$, since $\beta\prec q$ and $\omega\prec p$ implies that $\omega\beta\prec p$. We have also $\mathit{cont}_q(\omega_n) = \bigcup_{\omega_n\beta\prec q, \beta\neq\lambda} \{\mathit{first}(\beta)\} \subseteq \bigcup_{\omega\beta\prec p, \beta\neq\lambda} \{\mathit{first}(\beta)\} = \mathit{cont}_p(\omega)$, since $\omega_n\beta\prec q$ and $\omega_1\ldots\omega_{n-1}\prec p$ implies that $\omega\beta\prec p$.

$\subseteq$ By induction on n.

<u>n=1</u>: By definition $\mathit{cont}_p(\omega_1) = \bigcup_{\omega_1\beta\prec p, \beta\neq\lambda} \{\mathit{first}(\beta)\}$. Let $\omega_1\beta=\omega'=\omega_1'\ldots\omega_m'$ where $\omega_i'\prec q$ for $1\leq i\leq m$. Since $\omega'\neq\lambda$, $m\geq 1$. We consider the three possible relationships between ω_1 and ω_1'.

(i) If ω_1' is a nontrivial prefix of ω_1 then $m\geq 2$, and since $\omega_2'\neq\lambda$ (recalling that $\lambda\not\prec q$), we have $\mathit{first}(\omega_2') \in \mathit{cont}_q(\omega_1') \subseteq \mathit{cont}_q$. But also $\mathit{first}(\omega_2') \in \mathit{first}_q$ violating R3. Contradiction.

(ii) If $\omega_1=\omega_1'$ then, since $\beta\neq\lambda$, we have $m\geq 2$. Again $\omega_2'\neq\lambda$ so $\mathit{first}(\beta) = \mathit{first}(\omega_2') \in \mathit{first}_q$.

(iii) If ω_1 is a nontrivial prefix of ω_1' then $\mathit{first}(\beta) \in \mathit{cont}_q(\omega_1)$.

<u>n>1</u>: Assume the result for n−1. If $\omega=\omega_1\ldots\omega_n$ then by definition $\mathit{cont}_p(\omega) = \bigcup_{\omega\beta\prec p, \beta\neq\lambda} \{\mathit{first}(\beta)\}$. If $\omega\beta\prec p$ then let $\omega\beta=\omega'=\omega_1'\ldots\omega_m'$ where $\omega_i'\prec q$ for $1\leq i\leq m$. Apply Lemma 7 as follows. We have ω is a prefix of ω'. Decompose ω as

$\omega = \omega_1\ \omega_2 \ldots \omega_n$ where $\omega_1 \propto q$ and $\omega_2 \ldots \omega_n \propto p$. Likewise $\omega' = \omega_1'\ \omega_2' \ldots \omega_m'$ where $\omega_1' \propto q$ and $\omega_2' \ldots \omega_m' \propto p$. Since $first_p = first_q$ by Lemma 3c, we have $cont_q \cap first_p = \phi$ (using R3). So by Lemma 7, $\omega_1 = \omega_1'$. We now have $\omega_2 \ldots \omega_n \beta = \omega_2' \ldots \omega_m' \propto p$, so $first(\beta) \in cont_p(\omega_2 \ldots \omega_n)$. By induction we have $cont_p(\omega_2 \ldots \omega_n) = cont_q(\omega_n) \cup first_q$.∎

Our final lemma about the set *cont* is the counterpart of Lemma 4 for the set *first*.

Lemma 11: If p is a pattern, $cont_p$ contains only tokens (not $\sim$).

Proof: By induction on the definition of patterns. If $p = \lambda$, "d", or "d" $\sim$ then $cont_p = \phi$. If $p = qr$ then by Lemma 8 and induction. If $p = (q|r)$ then by Lemma 9, induction, and Lemma 4. If $p = (q)^*$ then by Lemma 10, induction, and Lemma 4.∎

The final two lemmas are proofs of the pattern utility programs LAMBDA-P and FIRST. Their correctness will follow almost directly from Lemmas 2 and 3.

Lemma 12 (LAMBDA-P): Let p be a pattern. Then (LAMBDA-P p) = T iff $\lambda \propto p$.

Proof: By induction on patterns. The program deals with five exclusive cases. When $p=\lambda$ the answer is T. When $p=$"d" or "d" $\sim$, then the answer is NIL. When $p=qr$ the answer is (AND (LAMBDA-P q) (LAMBDA-P r)), by induction and Lemma 2. Similarly, when $p=(q|r)$ the answer is (OR (LAMBDA-P q) (LAMBDA-P r)), and when $p=(q)^*$ the answer is T.∎

Lemma 13 (FIRST): Let p be a pattern. Then (FIRST p) = a list containing the symbols of $first_p$.

Proof: By induction on patterns, the same five exclusive cases as the previous lemma; we use now Lemma 3 inductively.. When $p=\lambda$ then NIL. When $p=$"d" or "d" $\sim$, then (d). When $p=qr$ then (APPEND (FIRST q) (COND((LAMBDA-P q) (FIRST r)))), where $\lambda \propto p$ is determined by Lemma 12. When $p=(q|r)$ then (APPEND (FIRST q) (FIRST r)). Finally, when $p=(q)^*$ then simply (FIRST q).∎

V.C Parse Theorem I

We present now the proof of the first PARSE theorem stated in Section V.A:

$$\forall e \in E'_D \; (P_D(W_D(e)) = e)$$

where E'_D is the set of expression trees defined formally in Section IV.B, W_D is the writing function of Section IV.B, and the parsing function P_D corresponds to the LISP program PARSE presented in Section IV.C. The program PARSE accepts as input a list of tokens; its value, if it halts error-free, is the LISP representation of an expression tree, as defined in Section IV.C. The final token in any input string to PARSE is the special termination symbol ⊣; the left binding power of this symbol is assumed to be -1, the only non-negative left binding power used. In terms of the program the theorem is:

<u>PARSE Theorem I</u>: If $e \in E'_D$ and $\delta = W_D(e)$ then

$$\text{(PARSE } -1 \text{ } (\delta \dashv)) = (\text{repr}[e] \dashv)$$

Its inductive proof requires a restatement in the following more general form:

<u>Theorem I.9</u>: If for some e and rbp we are given

C1. $e \in E'_D$ and $\delta = t_1 \ldots t_k = W_D(e)$ for $k \geq 1$.

C2. $\text{r-index}[e] \geq \text{lbp}[t_{k+1}]$.

C3. $t_{k+1} \notin \text{c-set}[e]$.

C4. $\text{rbp} < \text{l-index}[e]$.

C5. $\text{rbp} \geq \text{lbp}[t_{k+1}]$.

then $$\text{(PARSE rbp } (\delta \; t_{k+1} \ldots)) = (\text{repr}[e] \; t_{k+1} \ldots).$$

PARSE Theorem I is a special case of Theorem 1.9 by the following argument. Cl is the given, letting $\delta = t_1 \ldots t_k$. The symbol t_{k+1} is ⊣ which has a left binding power of -1; from Lemma I we know that r-index[e] $\geq$ Ø, so C2 is satisfied. For condition C3 we observe that since ⊣ is not is the defined language, it cannot be in c-set[e]. As above, we know that l-index[e] $\geq$ Ø, so C4 is true. Finally, we know that rbp = lbp[⊣] = −1, satisfying C5.

Outline of Proof

Theorem 1.9 is the last in a sequence of nine subsidiary theorems, which correspond roughly to the subroutines of the program PARSE. Theorem 1.1 (FIND) covers the correct parsing of the annotation part of an expression. Theorems 1.2, 1.3, and 1.4 (NILFIX, PREFIX, and NUL-TYPE) deal with NUL-TYPE operators, and Theorems 1.5, 1.6, and 1.7 (POSTFIX, INFIX, and LEF-TYPE) similarly treat LEF-TYPE operators. Theorems 1.8 and 1.9 (PARSEa and PARSEb) state the top level behavior of the PARSE and ASSOC programs, the essential part of the parsing algorithm; Theorem 1.8 corresponds to the recursive parsing of left arguments and Theorem 1.9 to right arguments. Each theorem guarantees that if its arguments meet certain conditions, then the result of the corresponding subroutine has the desired property; i.e., that the subroutine operates correctly. With the exception of the language definition attached to property lists, as described in Section IV.C, each subroutine uses only values given as explicit arguments. No side-effects need be mentioned since the given implementation of PARSE contains only local variables.

The theorems are proven using simultaneous induction over the set E'_D of expression trees. At each level of induction, they may be proven sequentially according to their dependence by subroutine calls, as diagrammed in the partial ordering of Figure 2. In this figure the proof of the upper theorem of a linked pair depends on the lower theorem; the inductive use of Theorems 1.8 and 1.9 is indicated at the bottom of the graph. For instance, Theorem 1.4 depends on Theorems 1.2 and 1.3, which in turn depend on 1.1. In addition, 1.3 and 1.1 depend inductively on 1.9.

We use simple induction in this theorem to correspond exactly to the definition of the domain E_D; i.e. using a basis and an induction step. This form of definition was chosen for clarity and precision. The nature of the domain would, however, allow a proof by strong induction (without a basis step), since the theorem only requires induction in the cases when there exist non-null subtrees. Rather than redefine the domain or create unnecessary

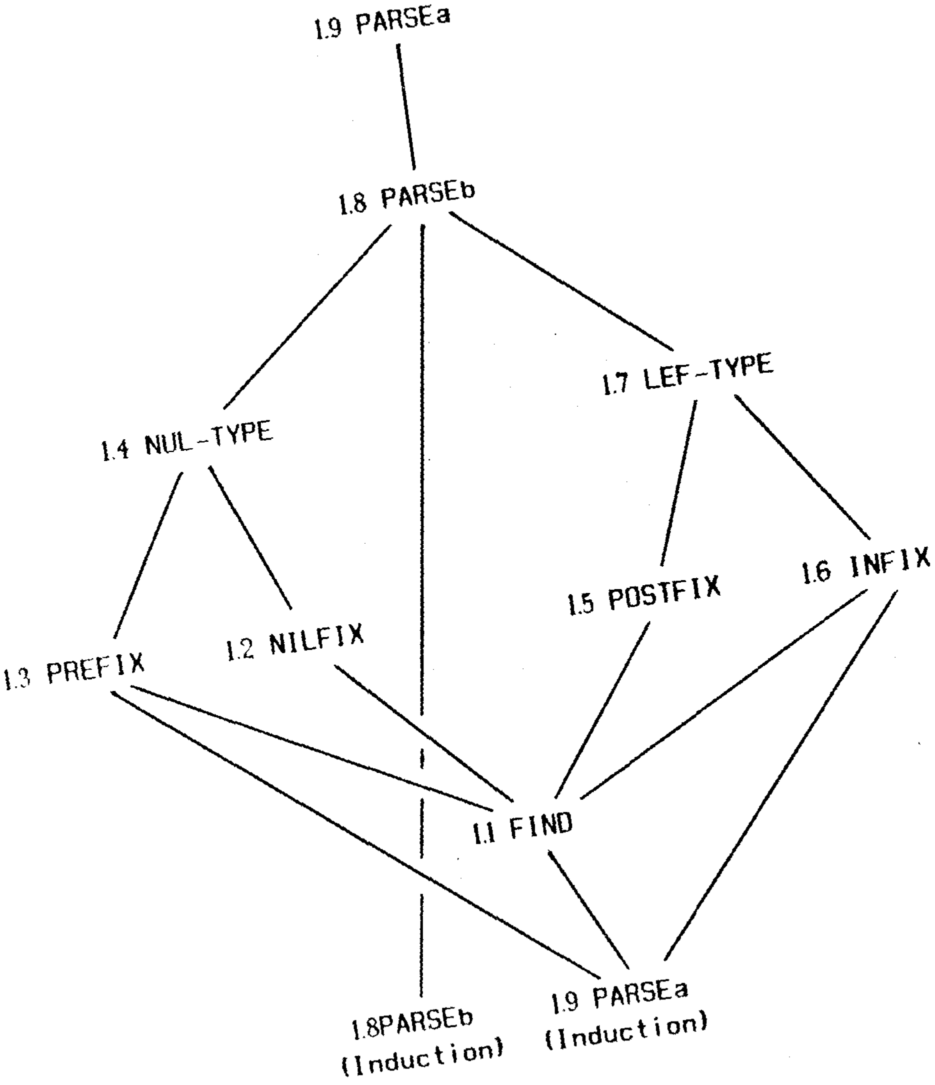

Figure 2. Interdependence of Theorems 1.1-1.9

confusion, strong induction is not used.

We now want to examine the five conditions we will impose on our input string in order to guarantee that PARSE returns the correct value. When viewed relative to a call to PARSE, they have the following interpretations. Condition 1 requires that the input string begin with a sentence δ of the language. Condition 2 insures that no subexpression on the right end of δ becomes associated as a left argument to t_{k+1}, if t_{k+1} is an operator. If t_{k+1} is a delimiter then condition 3 prevents its inclusion in any annotation within δ. The right binding power of the call to PARSE must be low enough for the entire expression to be returned, condition 4, but not so low that the expression is given as a left argument to t_{k+1}, condition 5.

Statement of Theorems 1.1 through 1.9

We precede our list of nine theorems by a formal statement of the conditions C1 through C5, on which they depend. For convenience in the proofs, the first three have been broken down into their definitional components: conditions C1a through C1f are equivalent to C1, C2a and C2b are equivalent to C2, and C3a and C3b are equivalent to C3.

Conditions:

C1. $e \in E'_D$ and $\delta = \alpha OP \beta \omega = W_D(e) = t_1 \ldots t_k$

- C1a. $\alpha = W_D(e_{left})$ if e_{left} exists (λ otherwise), $\beta = W_D(e_0)$ if e_0 exists (λ otherwise), and $\omega = d_1\gamma_1 \ldots d_n\gamma_n$ for $n \geq 0$, where $\gamma_i = W_D(e_i)$ for $1 \leq i \leq n$ when e_i is non-null (λ otherwise), and $e_{left}, e_0, e_1, \ldots e_n \in E'_D$ when they exist and are non-null.
- C1b. $\text{r-index}[e_{left}] \geq \text{lbp}[op]$ if e_{left} exists.
- C1c. $\text{rbp}[op] < \text{l-index}[e_0]$ if e_0 exists.
- C1d. $\text{rbp}[op] < \text{l-index}[e_i]$ for $1 \leq i \leq n$, when e_i is non-null.
- C1e. $d_1 \notin \text{c-set}[e_0]$ if e_0 and d_1 exist.
- C1f. $d_i \notin \text{c-set}[e_{i-1}]$ for $1 < i \leq n$ when e_i is non-null.

C2. $\text{r-index}[e] \geq \text{lbp}[t_{k+1}]$.

- C2a. $\text{rbp}[op] \geq \text{lbp}[t_{k+1}]$ if e_n exists and is non-null.
- C2b. $\text{r-index}[e_n] \geq \text{lbp}[t_{k+1}]$ if e_n exists and is non-null.

C3. $t_{k+1} \notin$ c-set[e].

C3a. $t_{k+1} \notin$ $cont_{p[op]}(e_1, \ldots e_n)$.

C3b. $t_{k+1} \notin$ c-set[e_n] if e_n exists.

C4. rbp < l-index[e].

C5. rbp ≥ lbp[t_{k+1}].

Notation:

(i) When writing LISP expressions, upper case words and parentheses will always refer to LISP code; when describing known values within LISP expressions, lower case and square brackets will be used. Specifically, the meta-variable *op* represents the token defined for *op*.

(ii) The representation of the annotation part produced by FIND is ((d$_1$ repr[e$_1$])...(d$_n$ repr[e$_n$])) and will be written repr[e$_1$,...e$_n$].

(iii) Since the representations of patterns are not manipulated in this program, we will abbreviate repr[*p*[*op*]] to simply *p*[*op*].

(iv) In proofs, we will use the names C1, C2, etc. to refer to the given conditions for the theorem being proved; C1', C2', etc. will refer to the antecedents to be satisfied when using Theorems 1.8 and 1.9 inductively.

We now state the nine theorems in full.

Theorem 1.1 (FIND): Given C1-C3 for some *e*. Then

(FIND rbp[*op*] (nil ω t_{k+1}...) *p*[*op*]) = (repr[e$_1$,...e$_n$] t_{k+1}...)

Theorem 1.2 (NILFIX): Given C1-C3 for some e. If *op* is defined NILFIX,

(NILFIX *op* (ω t_{k+1}...) rbp[*op*] *p*[*op*]) = (repr[e] t_{k+1}...)

Theorem 1.3 (PREFIX): Given C1-C3 for some e. If *op* is defined PREFIX,

(PREFIX *op* (β ω t_{k+1}...) rbp[*op*] *p*[*op*]) = (repr[e] t_{k+1}...)

Theorem 1.4 (NUL-TYPE): Given C1-C3 for some e. If *op* is defined NILFIX or PREFIX, then

$$(\text{NUL-TYPE } (op\ \beta\ \omega\ t_{k+1}\ldots)) = (\text{repr}[e]\ t_{k+1}\ldots)$$

Theorem 1.5 (POSTFIX): Given C1-C3 for some e. If *op* is defined POSTFIX,

$$(\text{POSTFIX repr}[e_{left}]\ op\ (\omega\ t_{k+1}\ldots)\ \text{rbp}[op]\ p[op]) = (\text{repr}[e]\ t_{k+1}\ldots)$$

Theorem 1.6 (INFIX): Given C1-C3 for some e. If *op* is defined INFIX,

$$(\text{INFIX repr}[e_{left}]\ op\ (\beta\ \omega\ t_{k+1}\ldots)\ \text{rbp}[op]\ p[op]) = (\text{repr}[e]\ t_{k+1}\ldots)$$

Theorem 1.7 (LEF-TYPE): Given C1-C3 for some e. If *op* is defined POSTFIX or INFIX, then

$$(\text{LEF-TYPE } (\text{repr}[e_{left}]\ op\ \beta\ \omega\ t_{k+1}\ldots)) = (\text{repr}[e]\ t_{k+1}\ldots)$$

Theorem 1.8 (PARSEa): Given C1-C4 for some e and rbp. Then

$$(\text{PARSE rbp } (\delta\ t_{k+1}\ldots)) = (\text{ASSOC rbp } (\text{repr}[e]\ t_{k+1}\ldots))$$

Theorem 1.9 (PARSEb): Given C1-C5 for some e and rbp. Then

$$(\text{PARSE rbp } (\delta\ t_{k+1}\ldots)) = (\text{repr}[e]\ t_{k+1}\ldots)$$

Proof of Theorems 1.1 through 1.9, Basis Step

For the basis step we assume that the tree $e \in E'_D$ is a single node whose label we denote *op*. Then *op* is defined NILFIX, $\lambda \propto p[op]$, and $t_1 = \delta = W_D(e) = op$ (n=0, k=1), so the annotation part is $\omega=\lambda$. Note that since *op* is defined NILFIX, Theorems 1.3, 1.5, 1.6, and 1.7 are not applicable.

Theorem 1.1 (FIND): If $\omega=\lambda$ matches $p[op]$ and if $t_i \not\propto cont_{p[op]}(\lambda)$ then

(FIND rbp[*op*] (nil t_2...) *p*[*op*]) = (nil t_2...)

Proof: The proof is by induction over the definition of the pattern *p*[*op*]; the six possible cases are handled by the six conditional clauses in the program.

Case 1. If *p*[*op*] = λ then (FIND rbp[*op*] (nil t_2...) *p*[*op*]) = (nil t_2...) immediately.

Cases 2,3. Impossible since if *p*[*op*] = "d" or "d" ~ it could not be that $\lambda \prec p[op]$.

Case 4. If *p*[*op*] = *qr*, then (FIND rbp[*op*] (nil t_2...) *p*[*op*])

= (FIND rbp[*op*] (FIND rbp (nil t_2...) *q*) *r*) by the program. We now use induction on the expression (FIND rbp (nil t_2...) *q*). Since $\lambda \prec p[op]$ and $t_2 \notin cont_{p[op]}(\lambda)$, we know by Lemma 2a that $\lambda \prec q$ and by Lemma 8b that $t_2 \notin cont_q(\lambda)$, so this expression is (nil t_2...) and we have

= (FIND rbp[*op*] (nil t_2...) *r*). As above we have $\lambda \prec r$ and $t_2 \notin cont_r(\lambda)$, so by another induction we have

= (nil t_2...).

Case 5. If *p*[*op*] = (*q*|*r*), then (FIND rbp[*op*] (nil t_2...) *p*[*op*]) is a conditional with three clauses. The first test is (MEMBER t_2 $first_q$), using Lemma 13 for the correctness of FIRST. Since $t_2 \notin cont_{p[op]}(\lambda)$, we know that $t_2 \notin first_q$ by Lemma 9a, and this test will fail. Similarly the second test (MEMBER t_2 $first_r$) will fail. The third test (LAMBDA-P *p*[*op*]) will be true by Lemma 12 and our assumption, so the result is (nil t_2...).

Case 6. If *p*[*op*] = (*q*)*, then (FIND rbp[*op*] (nil t_2...) *p*[*op*]) is a conditional with two clauses. The first test is (MEMBER t_2 $first_q$). Since $t_2 \notin cont_p(\lambda)$ we have by Lemma 10 that $t_2 \notin first_q$, and the test fails. The second clause then always returns (nil t_2...).∎

Theorem 1.2 (NILFIX): Given C1-C3 for some *e*. If *op* is defined NILFIX,

(NILFIX *op* (t_2...) rbp[*op*] *p*[*op*]) = (repr[e] t_2...).

Proof: From the program we have the expression

```
= (CONS(APPEND (op)
               (CAR(FIND rbp[op] (nil t2...) p[op])))
       (CDR(FIND rbp[op] (nil t2...) p[op])))
```

By Theorem 1.1 we know the call to FIND returns (nil t_2...), so we have
= (CONS (APPEND (*op*) nil) (t_2...)), which is
= (repr[e] t_2...) by the definition of representation.∎

<u>Theorem 1.4</u> (NUL-TYPE): Given C1-C3 for some *e*. If *op* is defined **NILFIX** or PREFIX, then

(NUL-TYPE (*op* t_2...)) = (repr[e] t_2...)

<u>Proof</u>: By the program and Axiom 1 covering definitions, we have
= (NILFIX *op* (t_2...) rbp[*op*] *p*[*op*]), which by Theorem 1.2 is
= (repr[e] t_2...).∎

<u>**Theorem 1.8**</u> (PARSEa): Given C1-C4 for some e and rbp. Then

(PARSE rbp (δ t_2...)) = (ASSOC rbp (repr[e] t_2...))

<u>Proof</u>: By the program we have
= (ASSOC rbp (NUL-TYPE (δ t_2...))), which by Theorem 1.4 is
= (ASSOC rbp (repr[e] t_2...)).∎

<u>**Theorem 1.9**</u> (PARSEb): Given C1-C5 for some e and rbp. Then

(PARSE rbp (δ t_2...)) = (repr[e] t_2...)

<u>Proof</u>: By Theorem 1.8 we have
= (ASSOC rbp (repr[e] t_2...)), which makes the test (LESSP rbp lbp[t_2]). By C5 this is false, so the value is
= (repr[e] t_2...).∎

<u>Proof of Theorems 1.1 through 1.9, Induction Step</u>

We assume that the tree $e \in E'_D$ is a node, whose label we denote *op*, with subtrees. We assume Theorems 1.8 and 1.9 inductively for any of these subtrees.

Theorem 1.1 (FIND): If $\omega = d_1\gamma_1 \ldots d_n\gamma_n$ matches $p[op]$ for $n \geq 0$ where $\gamma_i = W_D(e_i)$ and $e_i \in E'_D$ for $1 \leq i \leq n$, and if C1d, C1f, C2b, C3a, and C3b hold for ω, then

$$\text{(FIND rbp}[op]\ \text{(nil}\ \omega\ t_{k+1}\ldots)\ p[op]) = (\text{repr}[e_1,\ldots e_n]\ t_{k+1}\ldots)$$

Proof: Since FIND is called recursively, in general there will be some annotation fragment v, not necessarily nil, which has already been parsed at some previous stage of the execution. Thus, we will actually prove a more general assertion than that in the problem statement itself:

$$\text{(FIND rbp}[op]\ (v\ \omega\ t_{k+1}\ldots)\ p[op]) = (v \bullet \text{repr}[e_1,\ldots e_n]\ t_{k+1}),$$

where, for convenience, the result of appending two lists a and b is written a•b. As in the basis step, the proof is by induction over the definition of the pattern $p[op]$; the six possible cases are handled by the six separate clauses of the conditional statement.

Case 1. If $p[op] = \lambda$ then (FIND rbp$[op]$ $(v\ \omega\ t_{k+1}\ldots)$ $p[op]$)
$= (v\ \omega\ t_{k+1}\ldots)$. Since $\omega = \lambda$, then $n=0$ and $k=0$, so
$= (v\ t_1\ldots)$. But repr$[e_1,\ldots e_n]$ = nil so we have
$= (v \bullet \text{repr}[e_1,\ldots e_n]\ t_{k+1}\ldots)$.

Case 2. If $p[op]$ = "d" then we must have $\omega = d = t_1$. By the program then, since d matches t_1, (FIND rbp$[op]$ $(v\ \omega\ t_{k+1}\ldots)$ $p[op]$)
= (CONS (APPEND v ((d))) $(t_2\ldots)$)
$= (v \bullet ((d))\ t_2\ldots)$ which is
$= (v \bullet \text{repr}[e_1,\ldots e_n]\ t_{k+1}\ldots)$.

Case 3. If $p[op]$ = "d" ~ then we must have $\omega = d_1\gamma_1$ where $\gamma_1 = W_D(e_1)$. By the program we have (FIND rbp$[op]$ $(v\ \omega\ t_{k+1}\ldots)$ $p[op]$)
= (CONS (APPEND v (LIST(LIST d (CAR (PARSE rbp$[op]$ $(\gamma_1\ t_{k+1}\ldots)$)))))
(CDR (PARSE rbp$[op]$ $(\gamma_1\ t_{k+1}\ldots)$)))

We apply Theorem 1.9 inductively to (PARSE rbp$[op]$ $(\gamma_1\ t_{k+1}\ldots)$). C1' is satisfied by our assumption about ω. Since n=1, C2' and C3' are satisfied by C2b and C3b respectively. Finally, C4' and C5' are satisfied directly by C1d and C2a. We have then (PARSE rbp$[op]$ $(\gamma_1\ t_{k+1}\ldots)$) = (repr$[e_1]$ $t_{k+1}\ldots$), so our result is
= (CONS (APPEND v ((d repr$[e_1]$))) $(t_{k+1}\ldots)$)
$= (v \bullet \text{repr}[e_1,\ldots e_n]\ t_{k+1}\ldots)$.

Case 4. If $p[op] = qr$ then we must have $\omega = \omega_1\omega_2$ where $\omega_1 = t_1\ldots t_j = d_1\gamma_1\ldots d_m\gamma_m$ matches q, with $m \geq 0$ and $j \geq 0$, and

$\omega_2 = t_{j+1}\ldots t_k = d_{m+1}\gamma_{m+1}\ldots d_n\gamma_n$ matches *r*, with $n \geq m$ and $k \geq j$.
By the program we have (FIND rbp[*op*] (*v* ω t_{k+1}...) *p*[*op*])
= (FIND rbp[*op*] (FIND rbp[*op*] (*v* ω t_{k+1}...) *q*) *r*). We first apply our inductive assertion to the nested expression which is equivalent to
(FIND rbp[*op*] (*v* ω_1 t_{j+1}...) *q*). Conditions C1d' and C1f' about the internal properties of ω_1 follow directly from C1d and C1f respectively. C2b' through C3a' deal with the token t_{j+1} so we must deal separately with the cases where $\omega_2 \neq \lambda$ and $\omega_2 = \lambda$. If $\omega_2 \neq \lambda$, then t_{j+1} must be a delimiter by Lemma 11, so lbp[t_{k+1}] = 0, satisfying C2b' and C2a'. In this case t_{j+1} is also the delimiter d_{m+1}, so C3b' is satisfied by C1f. Finally, we know that $t_{j+1} \in \mathit{first}_r$, and by R1 $\mathit{cont}_q \cap \mathit{first}_r = \phi$, so C3a' is satisfied. If $\omega_2 = \lambda$, then $t_{j+1} = t_{k+1}$, and $m = n$. In this case, since $e_m = e_n$, C2b', C2a', and C3b' are satisfied directly by C2b, C2a, and C3b. Finally, since $t_{j+1} = t_{k+1} \notin \mathit{cont}_{p[op]}(e_1,\ldots e_n)$ by C3a, Lemma 8b says that $t_{j+1} \notin \mathit{cont}_q(e_1,\ldots e_n)$, satisfying C3a'. We have then by induction that this nested expression is (*v*@repr[$e_1,\ldots e_m$] t_{j+1}...), so we have
= (FIND rbp[*op*] (*v*@repr[$e_1,\ldots e_m$] ω_2 t_{k+1}...) *r*). We again use the assertion inductively. As before C1d' and C1f' are directly satisfied, but since the last part of ω_2 is also the last part of ω conditions C2b', C2a', and C3b' are also directly satisfied. Since $t_{k+1} \notin \mathit{cont}_{p[op]}(e_1,\ldots e_n)$, Lemma 8 says that $t_{k+1} \notin \mathit{cont}_r(e_{m+1},\ldots e_n)$, satisfying C3a'. We have finally,
= (*v*@repr[$e_1,\ldots e_m$]@repr[$e_{m+1},\ldots e_n$] t_{k+1}...)
= (*v*@repr[$e_1,\ldots e_n$] t_{k+1}...)

Case 5. If *p*[*op*]=(*q*|*r*) then by the program we have
(FIND rbp[*op*] (*v* ω t_{k+1}...) *p*[*op*])
= (COND ((MEMBER t_1 first_q) (FIND rbp[*op*] (*v* ω t_{k+1}...) *q*))
((MEMBER t_1 first_r) (FIND rbp[*op*] (*v* ω t_{k+1}...) *r*))
((LAMBDA-P *p*[*op*]) (*v* ω t_{k+1}...)))

It must be that either $\omega \neq \lambda$ matches *q*, $\omega \neq \lambda$ matches *r*, or $\omega = \lambda$. In the first case we have $t_1 \in \mathit{first}_q$, so the first test is true, and we get the value of
(FIND rbp[*op*] (*v* ω t_{k+1}...) *q*). All conditions for induction are satisfied automatically except C3a'. From C3a we have that $t_{k+1} \notin \mathit{cont}_{p[op]}(e_1,\ldots e_n)$, but then Lemma 9b tells us that $t_{k+1} \notin \mathit{cont}_q(e_1,\ldots e_n)$. By induction then, this returns the correct value. In the second case, the first test will fail because $t_1 \in \mathit{first}_r$, and R2 says that $\mathit{first}_q \cap \mathit{first}_r = \phi$. The second will be true, and as above the correct result will be returned. In the final case where $\omega = \lambda$, the first two tests must fail for the

following reason: C3a says that $t_{k+1} = t_1 \notin \mathit{cont}_{p[op]}(e_1,\ldots e_n) = \mathit{cont}_{p[op]}(\lambda)$, so we know by Lemma 9a that t_1 can be in neither first_q nor first_r. By Lemma 12 (LAMBDA-P $p[op]$) will be true, so (v ω $t_{k+1}\ldots$) is returned; since repr$[e_1,\ldots e_n]$ = nil, we here too get

= (v@repr$[e_1,\ldots e_n]$ t_{k+1}).

Case 6. If $p[op]$ = $(q)*$ then by the program (FIND rbp$[op]$ (v ω $t_{k+1}\ldots$) $p[op]$)

= (COND ((MEMBER t_1 first_q)
        (FIND rbp$[op]$ (FIND rbp$[op]$ (v ω $t_{k+1}\ldots$) q) $p[op]$))
      (T (v ω $t_{k+1}\ldots$)))

By definition either $\omega=\lambda$ or $\omega=\omega_1\ldots\omega_r$ for $r\geq 0$, where $\omega_i \propto q$ for $1\leq i\leq r$. If $\omega=\lambda$ then the first test must fail for the following reason: from C3a we have $t_1 = t_{k+1} \notin \mathit{cont}_{p[op]}(e_1,\ldots e_n) = \mathit{cont}_q(\lambda)$. But by Lemma 10a we know then $t_1 \notin \mathit{first}_q$., so the correct result is returned. For $r>0$ we prove the assertion by induction on r.

<u>n=0</u>. Then we have $\omega = \omega_1 = d_1\gamma_1\ldots d_n\gamma_n$ matches q. By the program we have

(FIND rbp$[op]$ (v ω_1 $t_{k+1}\ldots$) $p[op]$)

= (COND ((MEMBER t_1 first_q)
        (FIND rbp$[op]$ (FIND rbp$[op]$ (v ω_1 $t_{k+1}\ldots$) q) $p[op]$))
      (T (v ω_1 $t_{k+1}\ldots$)))

By our assumption the test $t_1 \in \mathit{first}_q$ will be true. We first apply our induction hypothesis on patterns to the nested expression (FIND rbp$[op]$ (v ω_1 $t_{k+1}\ldots$) q). We know by assumption that $\omega_1 \propto q$. Conditions C1d' through C3b' are satisfied directly by C1d through C3b respectively. From C3a we know that $t_{k+1} \notin \mathit{cont}_{p[op]}(e_1,\ldots e_n)$. By Lemma 10b we know that $t_{k+1} \notin \mathit{cont}_q(\omega_n)$, where $\omega_n=\omega_1$, so C3a' is satisfied. We then have

= (FIND rbp$[op]$ (v@repr$[e_1,\ldots e_n]$ $t_{k+1}\ldots$) $p[op]$). Now we know by C3a that $t_{k+1} \notin \mathit{cont}_{p[op]}(e_1,\ldots e_n)$ so we know by Lemma 10b that $t_{k+1} \notin \mathit{first}_q$. The test is false and the value of the program is

= (v@repr$[e_1,\ldots e_n]$ $t_{k+1}\ldots$).

<u>n>0</u>. Then we have $\omega = \omega_1\omega_2\ldots\omega_n$ where $\omega_1 = t_1\ldots t_j = d_1\gamma_1\ldots d_m\gamma_m$ matches q, with $m>0$ and $j>0$, and $\omega_2\ldots\omega_n = t_{j+1}\ldots t_k = d_{m+1}\gamma_{m+1}\ldots d_n\gamma_n$ matches $p[op]$, with $n>m$ and $k>j$. By the program we have (FIND rbp$[op]$ (v ω $t_{k+1}\ldots$) $p[op]$)

= (COND ((MEMBER t_1 first_q)
        (FIND rbp[*op*] (FIND rbp[*op*] (v ω t_{k+1}...) q) p[*op*]))
       (T (v ω t_{k+1}...)))

By our assumption the test $t_1 \in \mathit{first}_q$ will be true. We first apply our induction hypothesis on patterns to the the string ω_1 in the nested expression (FIND rbp[*op*] (v ω_1 $\omega_2 \ldots \omega_n$ t_{k+1}) q). Condition I1' is true by our assumption about ω. C1d' and C1f' are true directly from C1d and C1f. By Lemma 5, t_{j+1}, the first symbol in $\omega_2 \ldots \omega_n$ is a delimiter, and so lbp[t_{j+1}] = 0, satisfying C2b' and C2a'. We also know that $t_{j+1}=d_{m+1}$, so C3b' is satisfied by C1f. Finally, since $t_{j+1} \in \mathit{first}_q$ we know by restriction R3 that $t_{j+1} \notin \mathit{cont}_q$, satisfying C3a'. We have then the value

= (FIND rbp[*op*] (v@repr[$e_1, \ldots e_m$] $\omega_2 .. \omega_n$ t_{k+1}) p[*op*]). We now apply the induction on n to the string $\omega_2 \ldots \omega_n$. Conditions C1d' through C3a' are directly satisfied by C1d through C3a respectively, so we have the value

= (v@repr[$e_1, \ldots e_m$]@repr[$e_{m+1}, \ldots e_n$] t_{k+1}...) which is

= (v@repr[$e_1, \ldots e_n$] t_{k+1}...).▮

<u>**Theorem 1.2**</u> (NILFIX): Given C1-C3 for some e. If *op* is defined NILFIX,

(NILFIX *op* (ω t_{k+1}...) rbp[*op*] p[*op*]) = (repr[e] t_{k+1}...)

<u>Proof</u>: From the program we have the expression

= (CONS (APPEND (*op*)
                (CAR (FIND rbp[*op*] (nil ω t_{k+1}...) p[*op*])))
       (CDR (FIND rbp[*op*] (nil ω t_{k+1}...) p[*op*])))

By Theorem 1.1 we know that the call to FIND returns (repr[$e_1, \ldots e_n$] t_{k+1}...), so

= (CONS (APPEND (*op*) repr[$e_1, \ldots e_n$]) (t_{k+1}...))

= (repr[e] t_{k+1}...).▮

<u>**Theorem 1.3**</u> (PREFIX): Given C1-C3 for some *e*. If *op* is defined PREFIX,

```
(PREFIX op (β ω t_{k+1}...) rbp[op] p[op]) = (repr[e] t_{k+1}...)
```

<u>Proof</u>: From the program we have the expression

```
= (CONS(APPEND (op)
                 (LIST(LIST 'RIGHT (CAR (PARSE rbp[op] (β ω t_{k+1}...)))))
                 (CAR(FIND rbp[op]
                           (CONS NIL (CDR (PARSE rbp[op] (β ω t_{k+1}...))))
                           p[op]))))
        (CDR(FIND rbp[op]
                  (CONS NIL (CDR(PARSE rbp[op] (β ω t_{k+1}...))))
                  p[op])) )
```

Since $\beta = W_D(e_0)$ and $e_0 \in E'_D$, we know that if we can show our five conditions hold for e_0 then we can apply Theorem 1.9 inductively in order to obtain

```
        (PARSE rbp[op] (β ω t_{k+1}...)) = (repr[e_0] ω t_{k+1}...).
```

From C1a we obviously have C1' satisfied. C1c tells us that rbp[*op*] ≤ l-index[e_0], which immediately gives us C4'. For C2', C3', and C5' we must consider whether the annotation part ω is the null-string or not. If $\omega \neq \lambda$, then the first token of ω is the delimiter d_1, so C3' is satisfied by C1e. Since lbp[t_{j+1}] = 0, conditions C2' and C5' are also satisfied. If $\omega = \lambda$, then $n = 0$ and we immediately get C3' from C3b, C2' from C2b, and C5' from C2a. Thus, the value of the expression is

```
= (CONS (APPEND (op)
                 ((right repr[e_0]))
                 (CAR (FIND rbp[op] (nil ω t_{k+1}...) p[op]))))
        (CDR (FIND rbp[op] (nil ω t_{k+1}...) p[op])))
```

By Theorem 1.1 we know that the value of the call to FIND is (repr[e_1,...e_n] t_{k+1}...), so the value of the expression is

= (repr[e] t_{k+1}...).∎

<u>**Theorem 1.4**</u> (NUL-TYPE): Given C1-C3 for some *e*. If *op* is defined NILFIX or PREFIX, then

```
(NUL-TYPE (op β ω t_{k+1}...)) = (repr[e] t_{k+1}...)
```

Proof: We consider the two possible cases. If *op* is defined NILFIX then
(GET *op* 'NUL-TYP) = NILFIX by Axiom 1, so we have by the program and Axiom 1
= (NILFIX *op* (β ω t_{k+1}...) rbp[*op*] *p*[*op*]), which by Theorem 1.2 is
= (repr[e] t_{k+1}...). Similarly, if *op* is defined PREFIX the correct value is returned by the program, Axiom 2, and Theorem 1.3. If there is no NUL-TYPE definition for *op*, then the value is
= (NILFIX *op* (t_{k+1}...) $\emptyset$ λ), which is the default condition. In this case *op* is assumed to be nilfix with no arguments and null pattern, so by Theorem 1.2 the correct value is returned.▮

Theorem 1.5 (POSTFIX): Given C1-C3 for some e. If *op* is defined POSTFIX,

(POSTFIX repr[e_{left}] *op* (ω t_{k+1}...) rbp[*op*] *p*[*op*]) = (repr[e] t_{k+1}...)

Proof: By the program we have the value

```
= (CONS (APPEND (op)
                ((left repr[$e_{left}$]))
                (CAR (FIND rbp[op] (nil $\omega$ $t_{k+1}$...) p[op])))
        (CDR (FIND rbp[op] (nil $\omega$ $t_{k+1}$...) p[op])))
```

By Theorem 1.1, the call to FIND has the value (repr[e_1,...e_n] t_{k+1}...), so we have the expression

```
= (CONS (APPEND (op)
                ((left repr[$e_{left}$]))
                repr[$e_1$,...$e_n$])
        ($t_{k+1}$...))
= (repr[e] $t_{k+1}$...).▮
```

Theorem 1.6 (INFIX): Given C1-C3 for some e. If *op* is defined INFIX,

(INFIX repr[e_{left}] *op* (β ω t_{k+1}...) rbp[*op*] *p*[*op*]) = (repr[e] t_{k+1}...)

Proof: By the program we have the expression

```
= (CONS (APPEND (op)
                ((left repr[$e_{left}$]))
                (LIST(LIST 'RIGHT (CAR(PARSE rbp[op] ($\beta$ $\omega$ $t_{k+1}$...)))))
                (CAR(FIND rbp[op]
                          (CONS NIL (CDR(PARSE rbp[op] ($\beta$ $\omega$ $t_{k+1}$...))))
                          p[op])))
        (CDR(FIND rbp[op]
                  (CONS NIL (CDR(PARSE rbp[op]($\beta$ $\omega$ $t_{k+1}$...))))
                  p[op])) )
```

We use Theorem 1.9 inductively on the expression (PARSE rbp[*op*] (β ω t_{k+1}...)) in exactly the same manner as in the proof of Theorem 1.3 (PREFIX), yielding the value (repr[e_0] ω t_{k+1}...). We have then the expression

```
= (CONS (APPEND (op)
                ((left repr[$e_{left}$]))
                ((right repr[$e_0$]))
                (CAR (FIND rbp[op] (nil $\omega$ $t_{k+1}$...) p[op])))
        (CDR (FIND rbp[op] (nil $\omega$ $t_{k+1}$...) p[op])))
```

By Theorem 1.1 we know that the call to FIND returns the correct value, giving us

```
= (CONS (APPEND (op)
                ((left repr[$e_{left}$]))
                ((right repr[$e_0$]))
                 repr[$e_1$,...$e_n$])
        ($t_{k+1}$...))
= (repr[e] $t_{k+1}$...).∎
```

<u>Theorem 1.7</u> (LEF-TYPE): Given C1-C3 for some e. If *op* is defined POSTFIX or INFIX, then

$$\text{(LEF-TYPE (repr}[e_{left}]\ op\ \beta\ \omega\ t_{k+1}\ldots)) = \text{(repr}[e]\ t_{k+1}\ldots)$$

<u>Proof</u>: We consider the two possible cases. If *op* is defined POSTFIX then (GET *op* 'LEF-TYP) = POSTFIX by Axiom 3, so we have by the program and Axiom 3
= (POSTFIX repr[e_{left}] *op* (ω t_{k+1}...) rbp[*op*] *p*[*op*]), which by Theorem 1.5 is

= (repr[e] t_{k+1}...). Similarly, if *op* is defined INFIX the correct value is returned by the program, Axiom 4, and Theorem 1.6.▮

Theorem 1.8 (PARSEa): Given C1-C4 for some e and rbp. Then

(PARSE rbp (δ t_{k+1}...)) = (ASSOC rbp (repr[e] t_{k+1}...))

Proof: We consider the two possible cases: *op* is NUL-TYPE or LEF-TYPE.

Case 1. If *op* is defined NILFIX or PREFIX then we have $\alpha=\lambda$ in $\delta = \alpha op \beta\omega$, so $t_1 = op$. By the program we have (PARSE rbp (*op* β ω t_{k+1}..))
= (ASSOC rbp (NUL-TYPE (*op* β ω t_{k+1}...))), which by Theorem 1.4 is
= (ASSOC rbp (repr[e] t_{k+1}...)).

Case 2. If *op* is defined POSTFIX or INFIX then $\alpha\neq\lambda$. We apply Theorem 1.8 inductively to the expression (PARSE rbp (α *op* β ω t_{k+1}...)). From C1a we have α = $W_D(e_{left})$ where $e_{left} \in E'_D$, satisfying C1'. From C1b we have r-index[e_{left}] $\geq$ lbp[*op*], satisfying C2'. We do not allow LEF-TYPE operators to be used as delimiters, so since only delimiters can occur in c-set[e_{left}], C3' is trivially satisfied. From C4 we have rbp < l-index[e], and since l-index[e] = min[lbp[*op*], l-index[e_{left}]], we have rbp < l-index[e_{left}], satisfying C4'. By induction, then, we have (PARSE rbp (α *op* β ω t_{k+1}...))
= (ASSOC rbp (repr[e_{left}] *op* β ω t_{k+1}...)). The value of the call to ASSOC is a conditional whose first test is (LESSP rbp lbp[*op*]). By the same argument we used to satisfy C4' above, we have rbp < lbp[*op*], so the test is true and the result is
= (ASSOC rbp (LEF-TYPE (repr[e_{left}] *op* β ω t_{k+1}...)). By Theorem 1.7 this is
= (ASSOC rbp (repr[e] t_{k+1}...)).▮

Theorem 1.9 (PARSEb): Given C1-C5 for some e and rbp. Then

(PARSE rbp (δ t_{k+1}...)) = (repr[e] t_{k+1}...)

Proof: By Theorem 1.8 we have (PARSE rbp (δ t_{k+1}...))
= (ASSOC rbp (repr[e] t_{k+1}...)). The value of the call to ASSOC is a conditional whose first test is (LESSP rbp lbp[*op*]). By C5 this is false, so the second clause returns (repr[e] t_{k+1}...).▮

V.D PARSE Theorem II

We complete this chapter with the proof of the second PARSE theorem stated in Section V.A:

$$\forall \delta \in \Sigma^* \ (P_D(\delta) \text{ halts error-free} \Rightarrow \delta \in S_D)$$

where Σ^* is any string of tokens and S_D is the defined language as described in Section IV.B. The program PARSE is given as input a list of tokens; if it halts error-free then that string must be the linear representation of a grammatical expression tree. Notice that our work is simplified by the fact that we do not worry about the value returned by the program; this leads us to adopt the following convention.

Notation: We write (...) = (...) to mean that the LISP expression on the left evaluates error-free to the value on the right. The presence of LISP expressions whose value need not be discussed will be indicated by (...).

We can now restate our theorem in terms of the program PARSE as follows.

PARSE Theorem II: If $\delta \in \Sigma^*$ and if (PARSE -1 (δ ⊣)) = ((...) ⊣), then $\delta \in S_D$.

Outline of Proof

The statement and proof of this theorem closely parallel those of the first PARSE theorem. As before, our desired result is a corollary of the last in a series of nine subsidiary theorems, which correspond (in this case precisely) to the subroutines of the program PARSE. These theorems, however, are now in the converse form: whenever the subroutine returns a value certain properties are shown to be true about the input string. The proof is again by simultaneous induction with the theorems proven sequentially at each level. Their interdependence, including the inductive use of Theorem 2.9, is illustrated in Figure 3. The essential difference between the two PARSE theorems is the domain of induction; in this case we use induction on the length of strings in the set Σ^*.

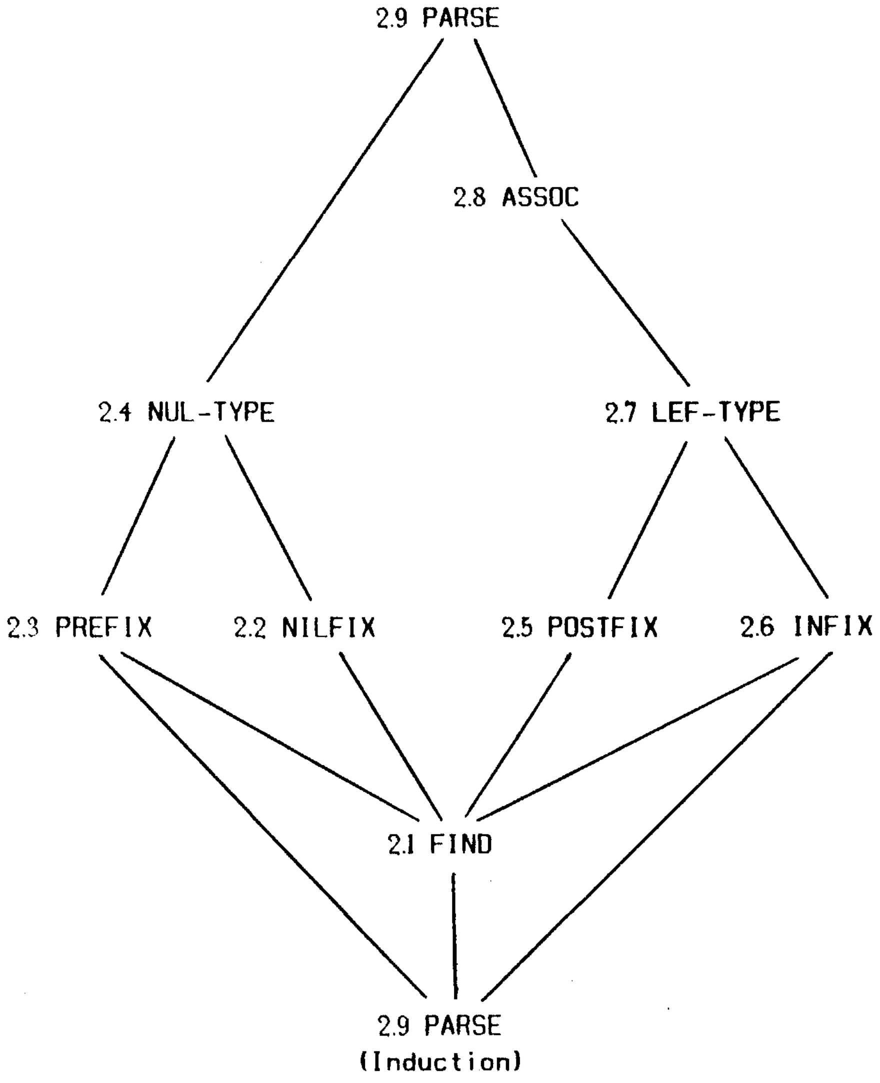

Figure 3. Interdependence of Theorems 2.1-2.9

Statement of Theorems 2.1 through 2.9

Since we are given that $\delta \in \Sigma^*$, we will assume that the input list to PARSE is the list of tokens $(t_1 \ldots t_s)$, for $s \geq 1$, with the convention that $t_s = \dashv$. As in the proof of PARSE Theorem I, we use an inductive generalization, Theorem 2.9, which makes use of Conditions C1 through C5. In this case, however, C1 through C5 are the consequents of the theorem. From C1 we have the desired result that $t_1 \ldots t_k \in S_D$.

Conditions:

C1. $e \in E'_D$ and $\delta = \alpha OP \beta \omega = W_D(e) = t_1 \ldots t_k$

C1a. $\alpha = W_D(e_{left})$ if e_{left} exists (λ otherwise), $\beta = W_D(e_0)$ if e_0 exists (λ otherwise), and $\omega = d_1\gamma_1 \ldots d_n\gamma_n$ for $n \geq 0$, where $\gamma_i = W_D(e_i)$ for $1 \leq i \leq n$ when e_i is non-null (λ otherwise), and $e_{left}, e_0, e_1, \ldots e_n \in E'_D$ when they exist and are non-null.

C1b. $\text{r-index}[e_{left}] \geq \text{lbp}[\mathit{op}]$ if e_{left} exists.

C1c. $\text{rbp}[\mathit{op}] < \text{l-index}[e_0]$ if e_0 exists.

C1d. $\text{rbp}[\mathit{op}] < \text{l-index}[e_i]$ for $1 \leq i \leq n$, when e_i is non-null.

C1e. $d_1 \notin \text{c-set}[e_0]$ if e_0 and d_1 exist.

C1f. $d_i \notin \text{c-set}[e_{i-1}]$ for $1 < i \leq n$ when e_i is non-null.

C2. $\text{r-index}[e] \geq \text{lbp}[t_{k+1}]$.

C2a. $\text{rbp}[\mathit{op}] \geq \text{lbp}[t_{k+1}]$ if e_n exists and is non-null.

C2b. $\text{r-index}[e_n] \geq \text{lbp}[t_{k+1}]$ if e_n exists and is non-null.

C3. $t_{k+1} \notin \text{c-set}[e]$.

C3a. $t_{k+1} \notin \mathit{cont}_{p[op]}(e_1, \ldots e_n)$.

C3b. $t_{k+1} \notin \text{c-set}[e_n]$ if e_n exists.

C4. $\text{rbp} < \text{l-index}[e]$.

C5. $\text{rbp} \geq \text{lbp}[t_{k+1}]$.

We state now the theorems in full.

<u>Theorem 2.1</u> (FIND): If (FIND rbp[*op*] (nil $t_{j+1} \ldots t_s$) *p*[*op*]) = state for $1 \le j < s$ then

(a) state = ((. . .) $t_{k+1} \ldots t_s$) where $j \le k < s$

(b) $t_{j+1} \ldots t_k = \omega = d_1\gamma_1 \ldots d_n\gamma_n$ for $n \ge 0$ where $\gamma_i = W_D(e_i)$ and $e_i \in E'_D$ for $1 \le i \le n$, and $\omega \propto p$.

(c) C1d, C1f, C2a, C2b, C3a, and C3b hold for $t_{j+1} \ldots t_k$.

<u>Theorem 2.2</u> (NILFIX): If t_1 = *op* is defined NILFIX and (NILFIX *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*]) = state, for $1 < s$, then

(a) state = ((. . .) $t_{k+1} \ldots t_s$) where $1 \le k < s$

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$.

<u>Theorem 2.3</u> (PREFIX): If t_1 = *op* is defined PREFIX and (PREFIX *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*]) = state, for $1 < s$, then

(a) state = ((. . .) $t_{k+1} \ldots t_s$) where $1 \le k < s$

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$.

<u>Theorem 2.4</u> (NUL-TYPE): If for $1 < s$ (NUL-TYP ($t_1 \ldots t_s$)) = state then

(a) state = ((. . .) $t_{k+1} \ldots t_s$) where $1 \le k < s$

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$ where t_1 is defined NILFIX or PREFIX.

<u>Theorem 2.5</u> (POSTFIX): If t_{a+1} = *op* is defined POSTFIX, $t_1 \ldots t_a = W_D(e_{left})$ for $e_{left} \in E'_D$, r-index[e_{left}] > lbp[t_{a+1}], and (POSTFIX (...) *op* ($t_{a+2} \ldots t_s$) rbp[*op*] *p*[*op*] = state where a+1<s then

(a) state = ((...) $t_{k+1} \ldots t_s$) where a<k<s

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$.

<u>Theorem 2.6</u> (INFIX): If t_{a+1} = *op* is defined INFIX, $t_1 \ldots t_a = W_D(e_{left})$ for $e_{left} \in E'_D$, r-index[e_{left}] > lbp[t_{a+1}], and (INFIX (...) op ($t_{a+2} \ldots t_s$) rbp[op] *p*[*op*]) = state where a+1<s then

(a) state = ((...) $t_{k+1} \ldots t_s$) where a<k<s

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$.

<u>Theorem 2.7</u> (LEF-TYPE): If $t_1 \ldots t_a = W_D(e_{left})$ for $e_{left} \in E'_D$, r-index[e_{left}] > lbp[t_{a+1}], and (LEF-TYPE (...) ($t_{a+1} \ldots t_s$)) = state where a<s then

(a) state = ((...) $t_{k+1} \ldots t_s$) where a<k<s

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$ where t_{a+1} is defined POSTFIX or INFIX.

<u>Theorem 2.8</u> (ASSOC): If $t_1 \ldots t_j$ satisfy C1, C2, C3, and C4, and (ASSOC rbp ((...) $t_{j+1} \ldots t_s$) = state where j<s then

(1) if rbp ≥ lbp[t_{j+1}], state = ((...) $t_{j+1} \ldots t_s$).

(2) If rbp < lbp[t_{j+1}], then
(a) state = (ASSOC rbp ((...) $t_{k+1} \ldots t_s$)) where j<k<s
(b) C1, C2, C3, and C4 hold for $t_1 \ldots t_k$ where t_{j+1} is defined POSTFIX or INFIX.

<u>Theorem 2.9</u> (PARSE): If (PARSE rbp $(t_1 .. t_s)$) = state for $1<s$ then

(a) state = $((...)\ t_{k+1} ... t_s)$ where $1 \leq k < s$

(b) C1 through C5 hold for $t_1 ... t_k$.

Since this proof is essentially concerned with error handling, we precede the basis step with a preliminary lemma about the behavior of the PARSE program on trivially invalid inputs. The theorem itself deals with list arguments to PARSE of length two or more, and it is important to know that no value will be returned for shorter lists.

<u>Lemma 2.1:</u> (PARSE rbp $(t_1 ... t_s)$) returns an error if $s<2$.

<u>Proof:</u> By the program (PARSE rbp $(t_1 ... t_s)$)
= (ASSOC rbp (NUL-TYPE $(t_1 ... t_s)$)), but NUL-TYPE immediately tests by evaluating (CDDR $(t_1 ... t_s)$). If $s<2$ this will cause an error.∎

<u>Proof of Theorems 2.1 through 2.9, Basis Step</u>

For the basis step we assume that $s=2$, so the input string is $(t_1\ t_2) = (t_1\ \dashv)$. Since Theorem 2.9 is the final result and is the only theorem to be used inductively, it is the only essential part of the basis step proof. To prove Theorem 2.9 for the case $s=2$ we will also need Theorems 2.1, 2.2, and 2.4.

<u>Theorem 2.1</u> (FIND): If (FIND rbp[op] (nil $\dashv$) $p[op]$) = state, then

(a) state = $((...)\ \dashv)$

(b) $\lambda = \omega$ matches $p[op]$

(c) C1d, C1f, C2a, C2b, C3a, and C3b hold for λ.

<u>Proof:</u> Since $k<s$, the second argument to FIND must be (nil $\dashv$). We prove then the following assertion inductively over the definition of the pattern $p[op]$. If

(FIND rbp[op] (nil ⊣) $p[op]$) = state, then $\lambda \prec p$ and state = (nil ⊣). This assertion implies that $\omega = \lambda$ and so C1d and C1f are trivially satisfied. Since lbp[⊣]=−1 C2 is satisfied, and C3 because ⊣ is not in the defined language.

Case 1. If $p[op]=\lambda$ then true immediately.

Cases 2,3. It cannot be that $p[op]$="d" or "d" ~, because a value would only be returned if d=⊣ and we know that ⊣ is not part of the defined language.

Case 4. If $p[op]=qr$ then the value must be (FIND rbp[op] (FIND rbp[op] (nil ⊣) q) r). By two uses of pattern induction we have $\lambda \prec q$ and $\lambda \prec r$ so $\lambda \prec p[op]$, and the final result (nil ⊣).

Case 5. If $p[op]=(q|r)$ then, since ⊣ cannot be in either of $first_q$ or $first_r$, it must be that $\lambda \prec p[op]$ and (nil ⊣) is returned.

Case 6. If $p[op]=(q)^*$ then, since ⊣ cannot be in $first_q$, the result (nil ⊣) is returned and $\lambda \prec p[op]$ by definition.▮

<u>Theorem 2.2</u> (NILFIX): If op is defined NILFIX and (NILFIX op (⊣) rbp[op] $p[op]$) = state then

(a) state = ((...) ⊣)

(b) C1, C2, and C3 hold for t_1.

<u>Proof:</u> By the program (NILFIX op (⊣) rbp[op] $p[op]$) = (CONS (...) (CDR(FIND rbp[op] (nil ⊣) $p[op]$))). By Theorem 2.1 this is = ((...) ⊣), and we know $\lambda \prec p[op]$. Since op is defined NILFIX we have then that $\delta = t_1 = op = W_D(e)$ for $e \in E'_D$, completing C1. We have already C2 and C3 from Theorem 2.1.

<u>Theorem 2.4</u> (NUL-TYPE): If (NUL-TYP (t_1 ⊣)) = state then

(a) state = ((...) ⊣)

(b) C1, C2, and C3 hold for t_1 where t_1 is defined NILFIX or PREFIX.

<u>Proof:</u> NUL-TYPE only returns a value in three cases.

Case 1. If $t_1 = op$ is defined NILFIX then we have the value (NILFIX op (⊣) rbp[op] $p[op]$) and we are done immediately by Theorem 2.2.

Case 2. If t_1=*op* is defined PREFIX then we have the value (PREFIX *op* (⊣) rbp[*op*] *p*[*op*]). But PREFIX evaluates the expression (PARSE rbp[*op*] (⊣)) which by Lemma 2.1 causes an error.∎

Case 3. If t_1=*op* is not defined, then it is assumed by default to be a variable or constant; we have then the value (NILFIX *op* (⊣) 0 λ) and we are again done by Theorem 2.2.

<u>Theorem 2.9</u> (PARSE): If (PARSE rbp (t_1 ⊣)) = state then

(a) state = ((...) ⊣)

(b) C1 through C5 hold for t_1.

<u>Proof:</u> By the program (PARSE rbp (t_1 ⊣)) = (ASSOC rbp (NUL-TYP (t_1 ⊣))). By Theorem 2.4 we know that this is = (ASSOC rbp ((...) ⊣)) and that C1, C2, and C3 hold. Since we know that lbp[⊣]=-1, ASSOC returns the value ((...) ⊣), and C5 is satisfied. Finally, since *op* has no left argument we have l-index[e] = ∞, satisfying C4.∎

<u>Proof of Theorems 2.1 through 2.9, Induction Step</u>

We now assume that s>2 and that Theorem 2.9 holds for strings of length less than s.

<u>Theorem 2.1</u> (FIND): If (FIND rbp[*op*] (nil $t_{j+1}\ldots t_s$) *p*[*op*]) = state for $1\leq j<s$ then

(a) state = ((...) $t_{k+1}\ldots t_s$) where $j\leq k<s$

(b) $t_{j+1}\ldots t_k = \omega = d_1\gamma_1\ldots d_n\gamma_n$ for $n\geq 0$ where $\gamma_i = W_D(e_i)$ and $e_i \in E'_D$ for $1\leq i\leq n$, and $\omega \propto p$.

(c) C1d, C1f, C2a, C2b, C3a, and C3b hold for $t_{j+1}\ldots t_k$.

<u>Proof:</u> The proof is by induction over the definition of the pattern *p*[*op*]; the six possible

cases are handled separately by the six conditional clauses in the program.

Case 1. If $p[op]=\lambda$ then (FIND rbp[op] (nil $t_{j+1}\ldots t_s$) $p[op]$))
= (($\ldots$) $t_{j+1}\ldots t_s$). In this case $\omega=\lambda$ which clearly matches $p[op]$. Only condition C3a is relevant to this case, but since $p=\lambda$, we have $cont_{p[op]} = \phi$.

Case 2. If $p[op]=$"d" then the program will only return a value if $d=t_{j+1}$. If it does, the value is
= (($\ldots$) $t_{j+2}\ldots t_s$). It must be the case that $j+1<s$, since $j+1=s$ would imply that $t_{j+1}=\dashv$ which we know cannot match any delimiter in the defined language. Clearly $\omega=t_{j+1}$ matches $p[op]$, and since there is no e_1, the only relevant condition to satisfy is again C3a. Since $p=$"d", we have $cont_{p[op]} = \phi$.

Case 3. If $p[op]=$"d" $\sim$, then the program will only return a value if $d=t_{j+1}$. If it does, the value is
= (($\ldots$) (CDR (PARSE rbp[op] ($t_{j+2}\ldots t_s$)))). We must have $j+2<s$, since PARSE returns an error otherwise by Lemma 2.1. We have then by an inductive use of Theorem 2.9 the value
= (($\ldots$) $t_{k+1}\ldots t_s$) where $j+2\leq k<s$. We also know that the following conditions hold for $\gamma_1 = t_{j+2}\ldots t_k$: C1' $\gamma_1=W_D(e_1)$ for $e_1\in E'_D$, C2' r-index[e_1] $\geq$ lbp[t_{k+1}], C3' $t_{k+1}\notin$ c-set[e_1], C4' rbp[op] $<$ l-index[e_1], and C5' rbp[op] $\geq$ lbp[t_{j+1}]. We now show that the necessary conditions hold for $t_{j+1}\ldots t_k$. We have first that $t_{j+1}\ldots t_k=d\gamma_1$ which clearly matches [op]. C1d is satisfied directly by C4'. Condition C1f does not apply, since $n=1$. C2a is satisfied by C5' and C2b by C2'. C3b is satisfied directly by C3' and C3a from the fact that $cont_{p[op]}=\phi$ in this case.

Case 4. If $p[op]=qr$, then the value of the program is
= (FIND rbp[op] (FIND rbp[op] (($\ldots$) $t_{j+1}\ldots t_s$) q) r). By pattern induction on the innermost expression we have the value
= (FIND rbp[op] (($\ldots$) $t_{h+1}\ldots t_s$) r) for $j\leq h<s$. We know that $t_{j+1}\ldots t_h=d_1\gamma_1\ldots d_m\gamma_m=\omega_1$, for $0\leq m$, which matches q, and that all conditions (call them C1d', C1f', etc.) hold for ω_1. By another use of pattern induction we have the value
= (($\ldots$) $t_{k+1}\ldots t_s$) for $h\leq k<s$. We know that $t_{h+1}\ldots t_k=d_{m+1}\gamma_{m+1}\ldots d_n\gamma_m=\omega_2$, for $m\leq n$, which matches r, and that all conditions (call them C1d", C1f", etc.) hold for ω_2. We now show that all conditions hold for $t_{j+1}\ldots t_k$. Clearly $j\leq k<s$ and $t_{j+1}\ldots t_k=\omega_1\omega_2$ matches $p[op]$. C1d follows directly from C1d' and C1d". C1f follows from C1f' and C1f" with one exception. We must show that $d_{m+1}\notin$ c-set[e_m]; i.e., that the first delimiter of ω_2 is not in the c-set of the last argument of ω_1. This case is covered by C3b'. If $\omega_2\neq\lambda$ then

C2a follows directly from C2a", otherwise from C2'. Similarly C2b follows from either C2b" or C2b'. We know from C3a" that $t_{k+1} \notin cont_r(e_{m+1}, \ldots e_n)$. If $\omega_2 \neq \lambda$ then by Lemma 8a we know $t_{k+1} \notin cont_{p[op]}(e_1, \ldots e_n)$. If $\omega_2 = \lambda$ then $t_{k+1} = t_{h+1}$ and we also know that $t_{j+1} \notin cont_q(e_1, \ldots e_m)$. But by Lemma 8b it is also true that $t_{k+1} \notin cont_{p[op]}(e_1, \ldots e_n)$, satisfying C3a. Finally, C3b follows from C3b" when $\omega_2 \neq \lambda$, otherwise from C3b'.

Case 5. If $p[op] = (q \mid r)$ then the program only returns a value in one of three cases. If the first test is true, $t_{j+1} \in first_q$, then we have

= (FIND rbp$[op]$ ((...) $t_{j+1} \ldots t_s$) q. By pattern induction this is

= ((...) $t_{k+1} \ldots t_s$), where all conditions except C3a are satisfied immediately. We know from C3a' that $t_{k+1} \notin cont_q(e_1, \ldots e_n)$. By Lemma 9b, since $\omega \neq \lambda$ in this case, we also know $t_{k+1} \notin cont_{p[op]}(e_1, \ldots e_n)$, satisfying C3a. If the first test is false and the second test, $t_{j+1} \in first_r$,is true then we have the same situation. If the first two tests are false, and the third is true, $\lambda \propto p[op]$, then the result is

= ((...) $t_{j+1} \ldots t_s$), where $\omega = \lambda$. In this case the only relevant condition is C3a. Since we know by the failure of the first two tests that $t_{j+1} \notin first_q$ and $t_{j+1} \notin first_r$, we know by Lemma 8 that $t_{j+1} \notin cont_{p[op]}(\lambda)$.

Case 6. If $p[op] = (q)*$ then value of the program is a conditional whose test is $t_{j+1} \in first_q$. If the test fails then the value is

= ((...) $t_{j+1} \ldots t_s$). If the test succeeds, then the value is a a recursive call to FIND for $p[op]$, after another ω_i has been found to match q. We prove by induction on the number of calls to FIND for $p[op]$ made before returning. The hypothesis is that each time there is a call of the form (FIND rbp$[op]$ ((...) $t_{h+1} \ldots t_s$) $p[op]$), then all of the conditions except C3a are true of the string $t_{j+1} \ldots t_h$. Thus, when the test finally fails, we only need show that C3a is true to be done, but by Lemma 10 we know that if $t_{k+1} \notin first_q$, when the test fails, and if $t_{k+1} \notin cont_q(e_n)$, which we know from the induction hypothesis C3b', then $t_{k+1} \notin cont_{p[op]}(e_1, \ldots e_n)$, satisfying C3a. We now prove the hypothesis.

<u>Basis</u>: At the first call, we have $j = h$, or $\omega = \lambda$. Since $p[op] = (q)*$, we know that $\omega \propto p[op]$. No other conditions are relevant to this case.

<u>Induction</u>: If all conditions except C3a are true of $t_{j+1} \ldots t_h = \omega_1 \ldots_{i-1}$ (call these conditions C1d', C1f', etc.), and if the test $t_{h+1} \in first_q$ is true, then we have

(FIND rbp$[op]$ ((...) ($t_{h+1} \ldots t_s$) $p[op]$)

= (FIND rbp$[op]$ (FIND rbp$[op]$ ((...) $t_{h+1} \ldots t_s$) q) $p[op]$). We know by our

induction over patterns (the higher level induction in this theorem) that this is
= (FIND rbp[*op*] ((...) $t_{k+1} \ldots t_s$) *p*[*op*]), where the string $t_{h+1} \ldots t_k = \omega_i$ matches q. We also know that all the conditions are true for this string (call these C1d", C1f", etc.). We now show that all conditions are true for the whole string $t_{j+1} \ldots t_{k+1}$. Since $\omega_1 \ldots \omega_{i-1} \prec p$ and $\omega_i \prec q$, we know that $t_{j+1} \ldots t_k = \omega_1 \ldots \omega_i \prec p$. C1d is satisfied directly by C1d' and C1d". C1f is similarly satisfied by C1f' and C1f" with one exception. We need to show that the first symbol of ω_n is not in the c-set of the last argument in $\omega_1 \ldots \omega_{i-1}$, but this follows from C3b'. We recall that Lemma 6 says that λ cannot match q. Then we have conditions C2a, C2b, and C3b following directly from C2a', C2b', and C3b' respectively. Thus all conditions except C3a are satisfied.∎

<u>Theorem 2.2</u> (NILFIX): If t_1 = *op* is defined NILFIX and (NILFIX *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*]) = state, for 1<s, then

(a) state = ((...) $t_{k+1} \ldots t_s$) where $1 \le k < s$

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$.

<u>Proof:</u> By the program (NILFIX *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*])
= (CONS (...) (CDR (FIND rbp[*op*] (nil $t_2 \ldots t_s$) *p*[*op*]))). So by Theorem 2.1
= ((...) $t_{k+1} \ldots t_s$) where $2 \le k < s$. Since the annotation part $t_2 \ldots t_k$ matches *p*[*op*] by the theorem and C1d and C1f hold, we have satisfied C1, because C1b, C1c, and C1e are not relevant to the NILFIX case. Then $t_1 \ldots t_k$ = $W_D(e)$ for $e \in E'_D$. By Theorem 2.1 we also have C2a, C2b, C3a, and C3b, which give us C2 and C3 for e.∎

<u>Theorem 2.3</u> (PREFIX): If t_1 = *op* is defined PREFIX and (PREFIX *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*]) = state, for 1<s, then

(a) state = ((...) $t_{k+1} \ldots t_s$) where $1 \le k < s$

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$.

<u>Proof:</u> By the program (PREFIX *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*])

= (CONS (...) (CDR (FIND rbp[*op*]
(CONS NIL (CDR (PARSE rbp[*op*] ($t_2 \ldots t_s$))))
p[*op*]))).

We first consider the expression (PARSE rbp[*op*] ($t_2 \ldots t_s$)). It must be the case that s>2, otherwise PARSE causes an error by Lemma 2.1. We have then by the inductive use of Theorem 2.9 that the result is

= (CONS (...) (CDR (FIND rbp[*op*] (nil $t_{j+1} \ldots t_s$) *p*[*op*]))), where we know the following about $t_2 \ldots t_j = \beta$: C1' $\beta = W_D(e_\theta)$ for $e_\theta \in E'_D$, C2' r-index[e_θ] ≥ lbp[t_{j+1}], C3' $t_{j+1} \notin$ c-set[e_θ], C4' rbp[*op*] < l-index[e_θ], and C5' rbp[*op*] ≥ lbp[t_{j+1}]. Finally, we know that j<s, so we apply Theorem 2.1 and get

= ((...) $t_{k+1} \ldots t_s$), where we know k<s and that C1d, C1f, C2a, C2b, C3a, and C3b already hold for the expression $t_1 \ldots t_k$. We satisfy the others as follows. We have now $t_1 \ldots t_k = op\beta\omega$ where *op* is defined PREFIX, and annotation part ω matches *p*[*op*], satisfying C1a. C1b is not relevant to this case. C1c is satisfied by C4'. C1e is only relevant if $\omega \neq \lambda$ in which case it is satisfied by C3'. If $\omega \neq \lambda$ then e_n is part of the annotation ω and conditions C2 and C3 follow from C2a, C2b, C3a, and C3b obtained from Theorem 2.1. If $\omega = \lambda$, then $e_n = e_\theta$. In this case C2 is satisfied by C5', and C3 is satisfied by C3a and C3'.∎

<u>Theorem 2.4</u> (NUL-TYPE): If for 1<s (NUL-TYP ($t_1 .. t_s$)) = state then

(a) state = ((...) $t_{k+1} \ldots t_s$) where 1≤k<s

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$ where t_1 is defined NILFIX or PREFIX.

<u>Proof:</u> NUL-TYPE returns a value in three possible cases.

Case 1. If $t_1 = op$ is defined NILFIX then we have the value

= (NILFIX *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*]) and the result is immediate by Theorem 2.2.

Case 2. If $t_1 = op$ is defined PREFIX then we have the value

= (PREFIX *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*]) and the result is immediate by Theorem 2.2.

Case 3. If t_1 is undefined, then it is assumed by default to be NILFIX with rbp=0 and *p*[*op*]=λ. We have the value

= (NILFIX *op* ($t_2 \ldots t_s$) 0 λ) and the result is immediate by Theorem 2.2.∎

Theorem 2.5 (POSTFIX): If $t_{a+1} = op$ is defined POSTFIX, $t_1 \ldots t_a = W_D(e_{left})$ for $e_{left} \in E'_D$, r-index$[e_{left}] > $ lbp$[t_{a+1}]$, and (POSTFIX (...) *op* ($t_{a+2} \ldots t_s$) rbp[*op*] *p*[*op*] = state where a+1<s then

(a) state = ((...) $t_{k+1} \ldots t_s$) where a<k<s

(b) Cl, C2, and C3 hold for $t_1 \ldots t_k$.

Proof: By the program we have (POSTFIX (...) *op* ($t_{a+2} \ldots t_s$) rbp[*op*] *p*[*op*])
= (CONS (...) (CDR (FIND rbp[*op*] (nil $t_{a+2} \ldots t_s$) *p*[*op*]))). So by Theorem 2.1
= ((...) $t_{k+1} \ldots t_s$) for $a+1 \leq k < s$, and $\omega = t_{a+2} \ldots t_k$ matches *p*[*op*] with Cld, Clf, C2a, C2b, C3a, and C3b already true. Since $t_1 \ldots t_a = \alpha = W_D(e_{left})$ by assumption we have $t_1 \ldots t_k = \alpha op \omega$, satisfying Cla. Clb is satisfied by given, and Clc and Cle are not relevant to the POSTFIX case, so we have $t_1 \ldots t_k = W_D(e)$ for $e \in E'_D$, satisfying Cl. C2 and C3 now follow directly from C2a, C2b, C3a, and C3b.∎

Theorem 2.6 (INFIX): If $t_{a+1} = op$ is defined INFIX, $t_1 \ldots t_a = W_D(e_{left})$ for $e_{left} \in E'_D$, r-index$[e_{left}] > $ lbp$[t_{a+1}]$, and (INFIX (...) *op* ($t_{a+2} \ldots t_s$) rbp[*op*] *p*[*op*]) = state where a+1<s then

(a) state = ((...) $t_{k+1} \ldots t_s$) where a<k<s

(b) Cl, C2, and C3 hold for $t_1 \ldots t_k$.

Proof: By the program we have(INFIX (...) *op* ($t_{a+2} \ldots t_s$) rbp[*op*] *p*[*op*])

```
= (CONS (...) (CDR (FIND rbp[op]
                         (CONS NIL (CDR (PARSE rbp[op] (ta+2...ts))))
                         p[op])))
```

Using the same argument as in Theorem 2.3, substituting ($t_{a+1} \ldots t_s$) for ($t_2 \ldots t_s$), we apply Theorems 2.9 and 2.1 in order to get
= ((...) $t_{k+1} \ldots t_s$). Conditions Cl-C3 are also satisfied for the same reasons as in Theorem 2.3, with the exception of Clb which is no longer irrelevant but is satisfied by the given.∎

Theorem 2.7 (LEF-TYPE): If $t_1 \ldots t_a$ = $W_D(e_{left})$ for $e_{left} \in E'_D$, r-index[e_{left}] > lbp[t_{a+1}], and (LEF-TYPE (...) ($t_{a+1} \ldots t_s$)) = state where a<s then

(a) state = ((...) $t_{k+1} \ldots t_s$) where a<k<s

(b) C1, C2, and C3 hold for $t_1 \ldots t_k$ where t_{a+1} is defined POSTFIX or INFIX.

Proof: It must be the case that a+1<s, otherwise LEF-TYPE returns an error by checking (CDDR ($t_{a+1} \ldots t_s$)), and LEF-TYPE only returns a value in following two cases.

Case 1. If t_{a+1}=*op* is defined POSTFIX then we have the value

= (POSTFIX (...) *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*]) and the result is immediate by Theorem 2.5.

Case 2. If t_{a+1}=*op* is defined INFIX then we have the value

= (INFIX (...) *op* ($t_2 \ldots t_s$) rbp[*op*] *p*[*op*]) and the result is immediate by Theorem 2.6.∎

Theorem 2.8 (ASSOC): If $t_1 \ldots t_j$ satisfy C1, C2, C3, and C4, and (ASSOC rbp ((...) $t_{j+1} \ldots t_s$) = state where j<s then

(1) if rbp ≥ lbp[t_{j+1}], state = ((...) $t_{j+1} \ldots t_s$).

(2) If rbp < lbp[t_{j+1}], then

(a) state = (ASSOC rbp ((...) $t_{k+1} \ldots t_s$)) where j<k<s

(b) C1, C2, C3, and C4 hold for $t_1 \ldots t_k$ where t_{j+1} is defined POSTFIX or INFIX.

Proof: The program is a conditional which tests (LESSP rbp lbp[t_{j+1}]). If the test is true then we have part 1. If false we have

= (ASSOC rbp (LEF-TYPE ((...) $t_{j+1} \ldots t_s$)). From the given we know C1', C2', C3', and C4' for $t_1 \ldots t_j$. By C1' and C2' the conditions for Theorem 2.7 are satisfied so we have

= (ASSOC rbp ((...) $t_{k+1} \ldots t_s$) where j<k<s. We know further that C1, C2, and C3 hold for $t_1 \ldots t_k$ where t_{j+1} is defined POSTFIX or INFIX. C4 holds since we have

rbp < lbp[t_{j+1}] by assumption and C4'.∎

Theorem 2.9 (PARSE): If (PARSE rbp (t_1..t_s)) = state for 1<s then

(a) state = ((...) t_{k+1}...t_s) where 1≤k<s

(b) C1 through C5 hold for t_1...t_k.

Proof: By the program (PARSE rbp (t_1...t_s))
= (ASSOC rbp (NUL-TYP (t_1.... t_s))). By Theorem 2.4 we know that this is
= (ASSOC rbp ((...) t_{j+1}...t_s) where 1≤j<s, and C1, C2, and C3 hold for t_1...t_j. Since t_1 is defined NILFIX or INFIX we know by definition that l-index[e] = ∞, so C4 is also satisfied. We know by Theorem 2.8 that ASSOC either returns when rbp[*op*] ≥ lbp[t_{j+1}] or calls itself recursively with conditions C1, C2, C3, and C4 still satisfied. By induction, when ASSOC does halt, C1, C2, C3, and C4 still hold. In addition condition C5 is satisfied by the failure of the test. Clearly ASSOC must eventually halt, since at each call we know j<k<s; i.e., every call removes more symbols from the input stream.∎

VI. CONCLUSIONS

VI.A Summary

We began with the observation that BNF is not effective as a practical meta-language for programming language designers, implementers, and users. We used Pratt's CGOL technique for translator construction, and specified a meta-language which avoids many of the difficulties inherent in BNF approaches. Its essential feature is an expressive power which is very closely related to the actual parsing technique of the translator: we can conveniently describe exactly those languages which the translator technique handles well. An immediate consequence is freedom from the awkward restrictions inherent in most automatic translator construction systems.

We have demonstrated these advantages by presenting the design of a CGOL based parsing program; although the meta-language is based on Pratt's informal syntactic guidelines, we have demonstrated with a formal correctness proof that none of the rigor of more traditional approaches has been sacrificed. The first part of this proof deals exclusively with properties of the meta-language; these results permit a very straightforward program proof, and may be applied equally well to proofs of other implementations.

VI.B Further Work

The use of nonstandard syntactic descriptions is an open area for research. The example presented in this paper treats a class of languages appropriate to the CGOL technique; it should be feasible to apply the same approach in other, perhaps more specialized, contexts. Even within the CGOL system there are a number of issues which need more thought. For example, the meta-language presented in this paper uses regular expressions to specify multiple right arguments. More than half of the proof is devoted to patterns, and the parser for them is the one long program in the system. The generality of regular expressions may not be worth the effort involved. Other unresolved issues deal with delimiters, e.g. it is not absolutely necessary that they have left binding powers of zero. This convention was imposed for simplicity.

There are also a number of unfinished implementation issues. The LISP implementation of the parser is much longer and less efficient than necessary but could be immediately improved by the use of global variables and side effects. The actual parser should be as short as most of the definitions for CGOL given in [Pratt 1974]. In addition, an actual implementation of the meta-language processor is desirable. This could take the form of an interactive definitional facility, providing the designer with on-line assistance, such as production debugging, and with incremental implementation, e.g. for bootstrapping.

SUMMARY OF NOTATION

p, q, r are CGOL annotation patterns

$\sim$ is a metasymbol used in productions to denote the presence of an argument

D is a language definition (a set of productions)

e is an expression tree

E_D is the set of expression trees corresponding to a definition D

E'_D is the set of grammatical expression trees corresponding to a definition D.

op is an operator

t is a token, a lexeme

d is a delimiting token

$\alpha, \beta, \gamma, \delta, \omega$ are strings of tokens

λ is the empty string

S_D is a set of strings, over some alphabet of tokens, corresponding to a definition D

W_D is a writing function defined on E_D with values in S_D

P_D is the parse function corresponding to a definition D

BIBLIOGRAPHY

[Aho 1968] A. V. Aho, "Indexed Grammars," *JACM*, 15:4 1968

[Aho, Johnson, & Ullman 1973] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Deterministic parsing of ambiguous grammars", *ACM Symposium on Principles of Programming Languages*, 1973.

[Baumann 1964] R. Baumann, M. Feliciano, F. L. Bauer, K. Samelson, *Introduction to ALGOL*, Prentice-Hall, Englewood Cliffs, N.J.

[Aho & Ullman 1972] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, N. J., 1972. *Volume 2: Compiling, 1973.*

[Cheatham 1967] T. E. Cheatham Jr., *The Theory and Construction of Compilers*, Computer Associates, Wakefield, Mass., 1967.

[Floyd 1963] R. W. Floyd, "Syntactic analysis and operator precedence". *JACM* 10:3 1963.

[Irons 1961] E. T. Irons, "A Syntax Directed Compiler for ALGOL 60", *CACM*, 4:1 1961.

[Knuth 1968] D. E. Knuth, "Semantics of context-free languages", *Math. Systems Theory*, 2:2 1968.

[Knuth 1971] D. E. Knuth, "Top-down syntax analysis". *Acta Informatica* 1:2 1971.

[Kurki-Suonio 1969] R. Kurki-Suonio, "Notes on Top Down languages," *BIT*, 9:3, 1969.

[Lewis & Stearns 1968] P. M. Lewis and R. E. Stearns, "Syntax directed transduction." *JACM* 15:3 1968.

[McCarthy 1963] J. McCarthy, "A basis for the mathematical theory of computation," *Computer Programming and Formal Systems*, North-Holland, Amsterdam, 1963.

[Nauer 1963] P. Nauer, "Revised report on the algorithmic language ALGOL 60," *CACM* 6:1, 1963.

[Pratt 1973] V. R. Pratt, "Top down operator precedence", *ACM Symposium on Principles of Programming Languages*, 1973.

[Pratt 1974] V. R. Pratt, "Quick and Clean - A Structured Approach to Language Definition," unpublished, March, 1974.

[Randell 1964] B. Randell and L. J. Russell, *ALGOL 60 Implementation*, Academic Press, New York, 1964.

[Rosenkrantz and Lewis 1970] D. J. Rosenkrantz and P. M. Lewis, "Deterministic left corner parsing", *IEEE Symposium on Switching and Automata Theory*, 1970.

[Stearns 1971] R. E. Stearns, "Deterministic top-down parsing". *Proc. Fifth Annual Princeton Conference on Information Sciences and Systems Theory*, 1971.