A PROGRAM DEVELOPMENT SYSTEM USING AN ATTRIBUTE GRAMMAR

by

KIRK BARRETT

B.S., Kansas State University, 1982

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements of the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, KS

1985

Approved by:

William J. Hankley
Major Professor

Table of Contents

THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

List of Figures and Tables

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. William Hankley for his advice and guidance which were essential to the completion of this report.

1. Introduction

A topic that concerns many people in the programming world is the "Software Crisis"— that is the need for and shortage of quickly-produced, correct, usable software. Software engineering's main goal has been to develop methods and tools to meet this need. Over the last ten to fifteen years, new methodologies and tools have been developed that have expedited the production and improved the quality of software: Warnier-Orr methodology and diagrams, specification languages, software life-cycle model and better programming languages are examples.

However, programming is still a difficult, complex task. There are many conceptual levels to be crossed between a formal specification of a problem and the final compilable program. If a programming system would allow the programmer to concentrate on the creative, more difficult and more abstract aspects of program development and let the more detailed, tedious, concrete aspects be automated or at least semi-automated, it would be valuable programming tool. In a such program development system (PDS) (also called program generators, program transformation systems or automatic programming systems), instead of actually entering program code character by character, the programmer would guide the system (or be guided by the system) through the program development process, and the actual code generation would be done by the system. If a system would produce programs which are generated, monitored, checked and/or corrected automatically (without any intervention) by the programmer or semi-automatically (with only a minimum of

intervention), it would be a tremendous tool in both expediting the production and ensuring the quality of software.

This report discusses program development systems. The most significant portion of the report is the discussion in section 4 of the prototype program development system (PPDS) that was developed. Section 5.2 in the appendix is a hard-copy listing of a sample terminal session using PPDS. The terminal session shows user responses to queries about the program to be developed (variable names, operations and structures) and the resultant Pascal program that was generated. The terminal session shows the usefulness of a PDS-- by answering simple queries about the type of program, a user can automatically generate a complete, compilable program. If the reader is solely interested in the approach and design of PPDS, then section 4 is of primary interest.

Section 2 presents an overview of PDS's in general-- theory, characteristics and a survey of PDS's that have appeared in literature. Section 3 is a tutorial on attribute grammars. One method of implementing a PDS is with an attribute grammar and it is the method PPDS uses. Sections 2 and/or 3 may be skipped if the reader is already familiar with or uninterested in their content.

2. Overview of Program Development Systems

2.1 Theory of PDS's

The fundamental problem in creating a PDS is to find a way to incorporate the knowledge of programming syntax, semantics and pragmatics into the system. Much is known about these areas, but, especially in pragmatics, most of this knowledge is intuitive and hard to encapsulate in precise rules or statements that are necessary for computer application. Expressing the knowledge needed in the three areas to develop programs is a "non-trivial" problem.

First, the knowledge of the syntax is needed to ensure a syntactically correct program is generated by the system. Fortunately, syntax can be precisely defined (for example, by BNF) and compilers that do have knowledge of the syntax of programming languages have been around since the late 50's. Structure editors, which ensure syntactically correct programs, also have been developed and used. They can be considered primative PDS's. Freeing the programmer from concentrating on syntactic details (which are easily overlooked or mistakenly done, yet are also conceptually simple and essential for a correct program), is a significant advance.

Next, the system must know the semantics of a language so that syntactic constructs can be combined in meaningful ways. Although this information is not as easily specified as the syntax, it is still fairly easily explained by and to humans, so it is not purely intuitive. Furthermore, efforts have been made to formally define

semantics through such methods as denotational semantics and attribute grammars.

To really represent a significant "break-through" in programming, however, (instead of just being a way to speed up traditional programming), a PDS must also include knowledge of programming pragmatics, that is, how the programming language constructs are used to solve real-world problems. As stated, pragmatic knowledge is highly intuitive and therefore hard to specify in precise, complete rules.

The following example illustrates the difference in the knowledge about syntax, semantics and pragmatics. The syntax of common programming language constructs such as assignment statements, "IF" statements and "WHILE" loops are easily precisely defined. The semantics of the three constructs are more abstract, but still easily understood by humans. When it comes to putting these constructs together, however, to create a payroll system, for example, it is very difficult to precisely explain how it is done.

Since the syntax and semantics of programming are not significant barriers to understanding and productivity, it is how well a PDS can incorporate knowledge about problem solving (for example, through cataloging previous problems solved, through encoding this information into a grammar or through a knowledge base of programming rules) that determines how useful, versatile, effective and powerful the PDS will be.

2.2 Characteristics of PDS's

In examing how programming knowledge can be incorporated, Barstow [BA79] identifies four characteristics or, as he calls them, approaches to automatic programming (or, as I call it, program development systems). The four characteristics that he lists are

- knowledge-based
- deductive
- high-level language
- transformational

The four characteristics in the list are in no way mutually exclusive, and, in fact, the boundaries between the four are not distinct. Nor is the list comprehensive: another characteristic, perhaps an even more significant one, is identified in this report. This fifth characteristic is whether a "general rule" or "target program domain" approach is used.

2.2.1 Knowledge-Based Approach

Systems using the first approach, knowledge-based systems, employ a large collection of rules representing knowledge about programming. The system will apply these rules in the development of the program. Almost every PDS uses some type of rule base, but how the rule base is actually implemented varies widely. In PPDS, programming knowledge is encoded in a rule base in the form of an attribute grammar (attribute grammars are discussed fully in section 3). The situation in which the rule base is separate from the actual PDS makes the rule base easily extended. PPDS is of this type. In other PDS's, the rule base is "hard-coded", which makes it not easily modified. The Cornell Program Synthesizer [TE80], a structure editor, has a hard-coded rule base.

2.2.2 Deductive Characteristic

If a system has the deductive characteristic, the PDS will try to "reason" or deduce what needs to be done in a program and what is the best way to do it. In PDS without deductive capabilities, the user will always initiate action and the system will merely respond. A deductive system will, however, take a much more active role in decisions about the development of a program. Of course, this makes the PDS more powerful and easier to use for a novice programmer, but requires much more sophistication to be effective. The DEDALUS (DEDuctive ALgorithm Ur-Synthesizer) system [MA78], which is described later, has the deductive characteristic; PPDS does not.

2.2.3 High-Level Language Characteristic

Another characteristic of PDS that Barstow lists is "high-level language". The significance that he sees in this characteristic is not clear, but it seems to mean that the system incorporates the use of a high-level programming language or a pseudo-code. The input to the system, the starting point of the development process, would be written in some form of this language. The systems which do not have this characteristic would start with an initial representation such as requirement specifications.

2.2.4 Transformational Approach

The final PDS approach listed is the transformational

approach. As the name implies, the attribute indicates that a program is developed by transforming more abstract, higher-level representations into more concrete, lower-level representations until finally actual compilable code is obtained. Again, almost all systems employ this approach, but how it's implemented varies widely. The high-level representation may start as something very far from code-- such as requirement specifications. Obviously, transforming these into code would require much work and sophistication. To make the transformation easier, a more code-like representation, such as a high-level pseudo- code or an abstract program template, may be chosen.

2.2.5 Target Program Domain Oriented vs General Rule Approach

As was mentioned earlier another characteristic or approach of PDS's which was not included in Barstow's list is whether the system is "general rule" or "target program domain" oriented. Systems of the first type contain the transformation rules which are relevant to any programming task and are "microscopic" (IE, because changes of small magnitude) so that any general program can be constructed from them. The particular domain of the program to be developed is irrelevant.

On the other hand, a target program domain oriented system's applicability is limited to certain problem domains. That is, the system can only develop programs which solve certain types of problems. In these systems, the transformational approach is still used, but the transformations are not as primitive or "microscopic"

as in the others and therefore, the system is not as versatile. While, of course, this limits the systems applicability, if the problem domain it does treat is large or common enough, the system can still be useful. Also, these systems are probably easier to use and more apt to develop a satisfactory program than the others. PPDS, as well as other attribute grammar PDS's, takes the target program domain approach.

This versatility-efficiency tradeoff is found quite often. A good analogy to illustrate this principle is the use of an adjustable wrench versus a socket wrench. Obviously, the adjustable wrench is more versatile: able to tighten nuts of many sizes. However, adjusting it to the right size is inconvenient and often this wrench will slip off because it is not quite adjusted correctly. The socket wrench, on the other hand, fits only one size nut, but does so perfectly. It can be used easily and efficiently. However, one is needed for every size nut to be used. The "general rule" systems are like the adjustable wrench: versatile, but perhaps a bit unwieldy, not quite fitting the problem at hand. The "target program domain" systems are like the socket wrench: not very versatile, but very well suited to the task for which they were made. If this task is common or important enough (eg., almost all the nuts to be tightened are the same size or development of many programs in the same general domain is needed), then this can be a very useful and practical tool.

2.3 Transformation Rules

Understanding transformation rules is a key to understanding PDS's according to Partsch and Steinbruggen [PA83]. They discuss program transformation systems, a name applicable to PDS's using the transformational approach. To them, a program transformation system, which could be used to either create new programs or optimize existing ones, is a system which supports a methodology of program construction by successive application of transformation rules. These rules govern how one program representation may be transformed into another. The rules ensure that the transformation is valid, so that the new representation is guaranteed correct. Depending on the system, the transformation may be selected automatically or chosen by the user.

Partsch and Steinbruggen state that the most common goal of a program transformation system is that of general support of program modification including optimization of control structures and selection of data structures. A different goal is that of program synthesis-- creating a program from an initial non-program representation such as a set of requirement specifications, mathematical assertions or restricted natural language. It is this second goal, program synthesis, which is of interest in this report.

2.4 Survey of Program Development Systems

2.4.1 Structure Editor: Cornell Program Synthesizer

The Cornell Program Synthesizer, CPS, is a syntax editor developed at Cornell University by Thomas Reps and Tim Tietlebaum.

It was used by students in a introductory programming course to develop PL/C programs. CPS ensures that a syntactically correct program is generated by displaying statement templates and verifying the information the user enters into them. This is a crude form of a PDS because, although CPS guarantees the the program is syntactically correct, it does not address semantic or pragmatic correctness at all. More advanced PDS's, including PPDS, do address semantic and pragmatic correctness.

2.4.2 General Rule Systems

Four transformational PDS's which were cited in the Partsch and Steinbruggen articleare summarized below. These systems can be identified as having the general rule characteristic. That is, the transformation rules the system applies are not dependent on the domain of the program to be developed. All these systems are written in a version of LISP. None of these systems use an attribute grammar and are not comparable to PPDS.

2.4.2.1 SAFE/TI/GIST

The SAFE/TI/GIST system was developed at the Information Science Institute by a team headed by Robert Balzer. It is written in INTERLISP and consists of three parts, as the name implies. The first part, SAFE (Specification Acquisition From Experts) [WI77], takes an informal description of a problem and its solution in a natural language and transforms this into a formal set of functional specifications. The next step of the system is the TI

(Transformational Implementor) [BA76]. It first transforms the functional specifications into algorithmic specifications written in the language GIST. Finally, the GIST specification is automatically translated into a programming language.

2.4.2.2 PSI/PMB/PECOS

Another PDS is the PSI/PMB/PECOS system [GR82], developed mostly at Stanford by C. Green and also written in LISP. The input to the PSI part of the system is a set of specifications derived by a user-system dialogue. These may be in natural language or input/output specifications. They are converted into program fragments and then passed to the Program Model Builder (PMB). At this stage, the fragments are transformed into an abstract algorithm in a pseudo-code using a base of 200 procedural rules. Next, the algorithm is sent to the "Coding Expert" called PECOS which, using a base of 400 coding rules, produces a executable LISP program.

2.4.2.3 Dershowitz and Manna

A third, unnamed system was developed by Dershowitz and Manna also at Stanford [DE77] and also written in LISP. Their system takes a unique approach: given a specification of a program to be constructed, the system attempts to find an analogy between this specification and the specification of a program which has already been constructed and cataloged. When such a program has been found, transformations are applied to the already existing program to produce a new program satisfying the new specifications. This

program is then presented to the user for any final touch-ups that may be needed.

2.4.2.4 DEDALUS

Finally, the DEDALUS system, which was also mentioned in the "deductive characteristic" section above, was developed at Stanford, by Manna and Waldinger [MA78]. It is implemented in QLISP and its input is high-level input/output specification in a LISP-like representation of math-logic notation. The system automatically and deductively derives a LISP program. The approach used is to achieve some goal expressed in the specifications by the use of meaning-preserving transformations until LISP code is produced.

2.4.3 Target Program Domain Systems

Two examples of the target program domain approach are the Cornell Program Synthesizer Generator (CPS-G) [RE84] and TRIAD [RA81].

2.4.3.1 CPS-G

CPS-G was developed at Cornell by Thomas Reps and Tim Tietlebaum. The target program domain for this system is structure editors modeled after the original CPS. Input into this system is the definition of the language for which a structure editor is desired. The definition is an attribute grammar specification which includes, to quote Reps, "rules defining abstract syntax, attribution, display format and concrete input syntax." An attribute grammar, which is also employed in the prototype PDS is this

report, is explained in detail later in the report.

2.4.3.2 TRIAD

The other system mentioned, TRIAD, was developed primarily by J. Ramanathan at Ohio State University. It was used to develop data processing software for Westinghouse Corporation and was shown to be capable of developing all of the 180 new programs written by Westinghouse over a period of 2 years. The basic methodology used was to provide the programmer with 4 fundamental algorithmic constructs from which many different programs in the business data processing domain could be developed. This system also employs an attribute grammar. PPDS takes much the same approach to program development as does the TRIAD system.

The use of an attribute grammar as the basis for a PDS seems to dictate that the system will be of the "target program domain" oriented type. This is because the system will be able to construct only the domain of programs defined in the grammar. Of course, if this domain becomes large enough, almost any program can be developed, and the distinction between the "general rule" and the "target program domain" approach is then no longer clear.

3. PDS's and Attribute Grammars

From the above discussion it is evident that attribute grammars have application in PDS's. Furthermore, PPDS presented in this paper employs an attribute grammar. For these two reasons, a section is included here that will first explain what an attribute grammar is (ie, what makes it different from an usual type of grammar), and then present a series of examples of increasing complexity and also of increasing practical interest starting with a simple translation grammar. Along the way, the reader should begin to grasp the idea of the attribute grammar and how it can be useful in a PDS. The examples will prepare the reader for the discussion of the prototype PDS and its attribute grammar, which appears later. If the reader is already familiar with attribute grammars, then skipping the section and moving to section 4 will cause no difficulty.

3.1 Definition of an Attribute Grammar

One major problem when working with an attribute grammar is the notation: it is difficult to construct a notation which is simple, consistent and meaningful in every case, and there certainly is not any standard in literature. I have tried to use a notation which meets the previous criteria. When I felt that a certain notation might be confusing, I have included an explanation. A standard BNF grammar definition is used, with a few exceptions. They are as follows:

1) a right arrow ("->") is used as a production symbol instead of "::="

2) semantic actions are enclosed in square brackets ("[" and "]") and are placed in a production at the point at which they should be executed

All the concepts of "normal" (that is, non-attribute) language grammars such as productions, derivation trees, nonterminal symbols and such also present in an attribute grammar. The distinction in an attribute grammar is that in addition to the syntactic rules and productions of the grammar, semantic rules and actions are also present . Semantic actions may range from the simple outputting of a symbol in a strict translation grammar to the construction of a code sequence in a PDS. In fact, this paper will cover these two extreme examples, which will illustrate just what is meant by a "semantic action".

3.2 Examples

3.2.1 Example 1: Translation Grammar

The following attribute grammar, taken directly from [LE76], illustrates the use of semantic actions. It translates infix arithmetic expressions to postfix notation. The semantic actions are simply the outputting of an appropriate symbol at the appropriate point in the translation. They are placed in the production at the point at which they should be "executed". Otherwise, standard BNF notation is used.

1. <E> -> <E> + <T> [OUTPUT('+')]

2. <E> -> <T>

3. <T> -> <T> * <P> [OUTPUT('*')]

4. <T> -> <P>

5. <P> -> (<E>)

6. <P> -> a [OUTPUT('a')]

7. <P> -> b [OUTPUT('b')]

8. <P> -> c [OUTPUT('c')]

Figure 1: Translation Grammar Example

In the example, when a certain production is chosen, the appropriate semantic action is carried along in the derivation of a string. When the parsing reaches a semantic action , it is executed which means to perform the "OUTPUT" function on the parameter. When an entire input string has been parsed, the translated string will have been output. A leftmost derivation of

(a+b) * c

would generate the tree

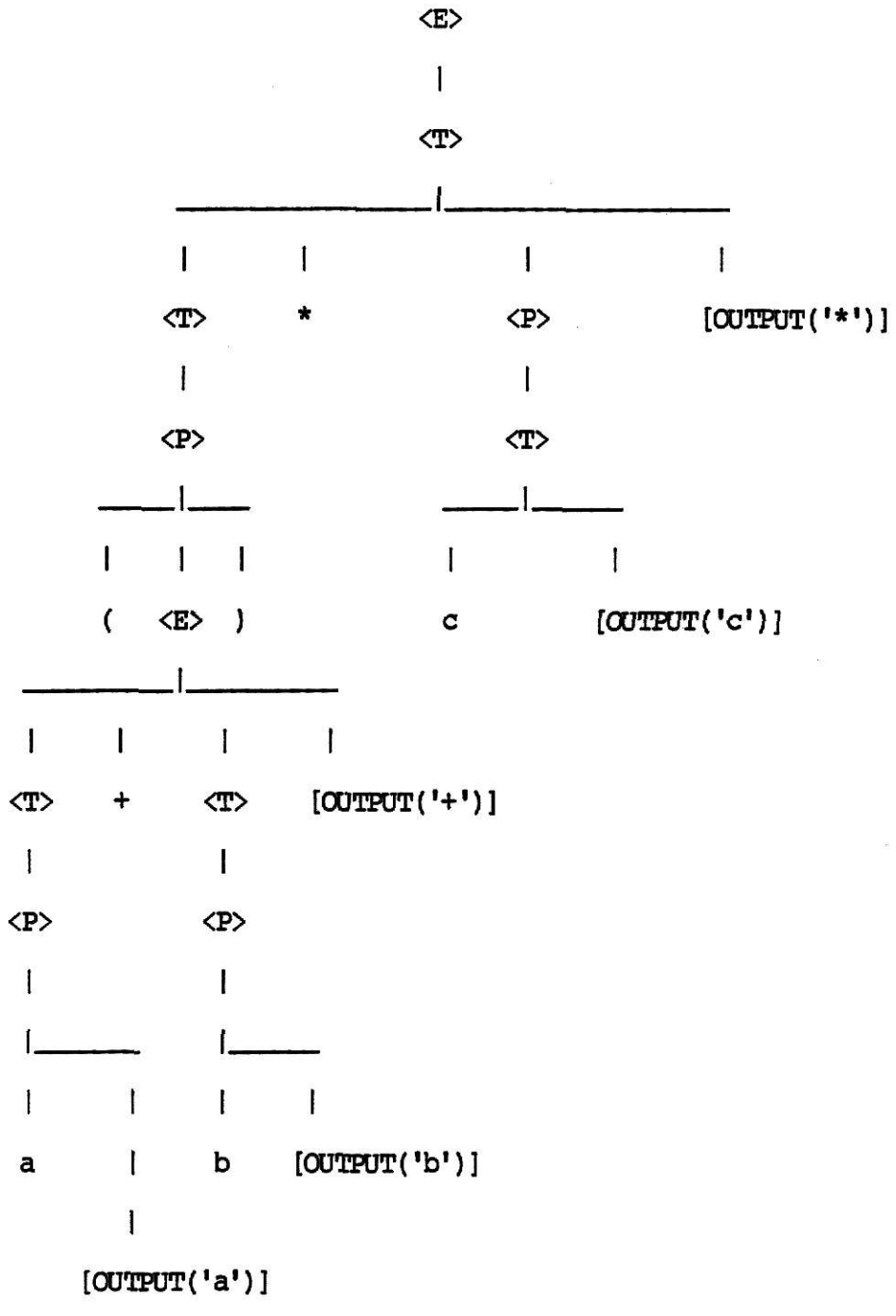


Figure 2: Translation Grammar Derivation Tree

From the leaf nodes of the tree, it can be seen that the output of this derivation would be

ab+c*

which is indeed the postfix equivalent of (a+b)*c.

3.2.2 Example 2: Synthesized Attributes

Although it is not apparent from the previous example, a basic idea behind an attribute grammar is that each node of a derivation tree has a value part associated with it (the value part of most of the above nodes was null). It is also a characteristic that the value parts may be determined from the value parts of other nodes: that is, the value parts or attributes may be passed up and down the tree. When the value part of the left-hand side of a production is determined from the value parts on the right-hand side, it is known as a "synthesized attribute". Another example, an expression evaluator, also taken from [LE76], illustrates the concept.

The notation is more difficult than in the first example. The same symbol is used for semantic actions and the value parts are represented in subscripts. Two new semantic action have been introduced, namely assignment and "VALUE".

1. <S> -> <E> [r <- q; OUTPUT(r)]

   q

2. <E> -> <E> + <T> [p <- q + r]

   p q r

3. <E> -> <T> [p<-q]

   p q

4. <T> -> <T> * <P> [p<-q*r]

   p q r

5. <P> -> <P> [p<- q]

   p q

6. <P> -> (<E>) [p <- q]

   r q

7. <P> -> c [p <- VALUE(c)] "c" is a constant

   p

Figure 3: Synthesized Attributes Grammar Example

The derivation is shown in two steps. In the first, the value parts are represented in the tree with symbols. In the completed derivation tree, all the value parts have been calculated.

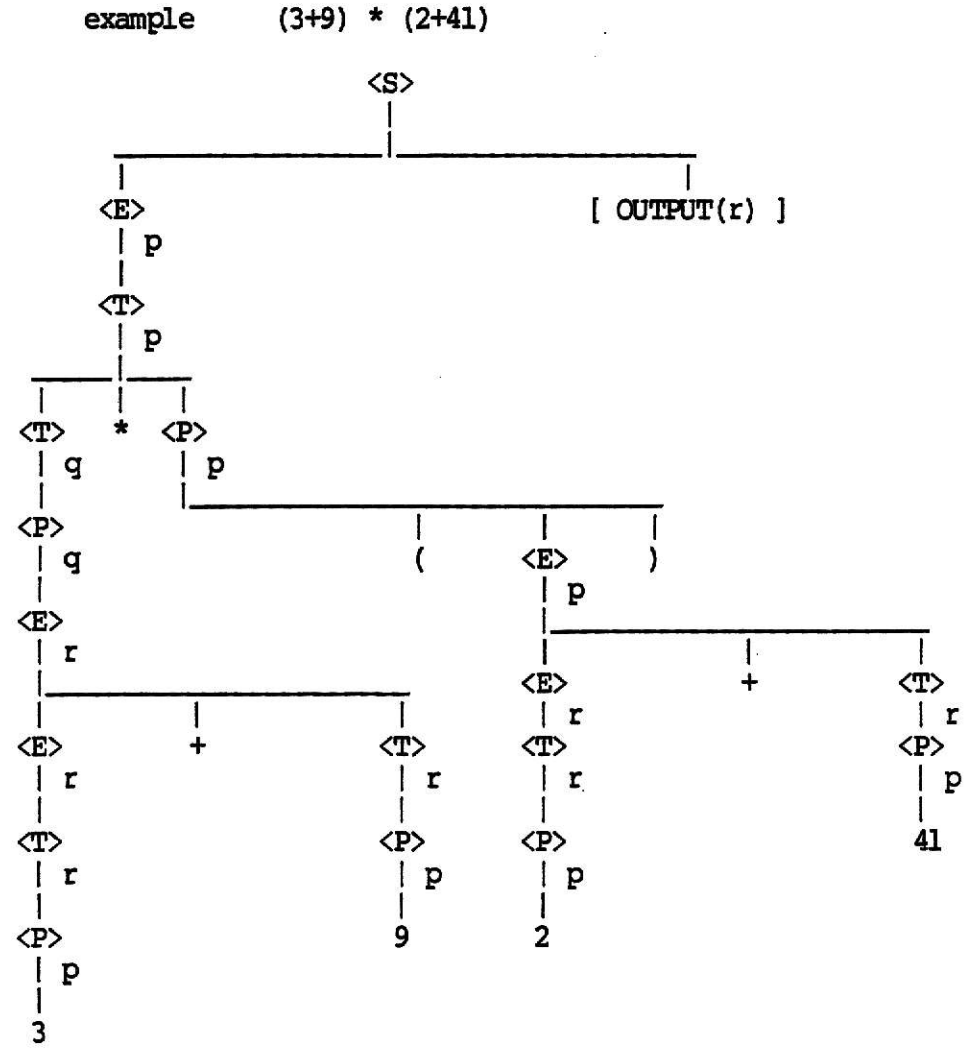


Figure 4: Synthesized Attributes Derivation Tree

The complete tree:

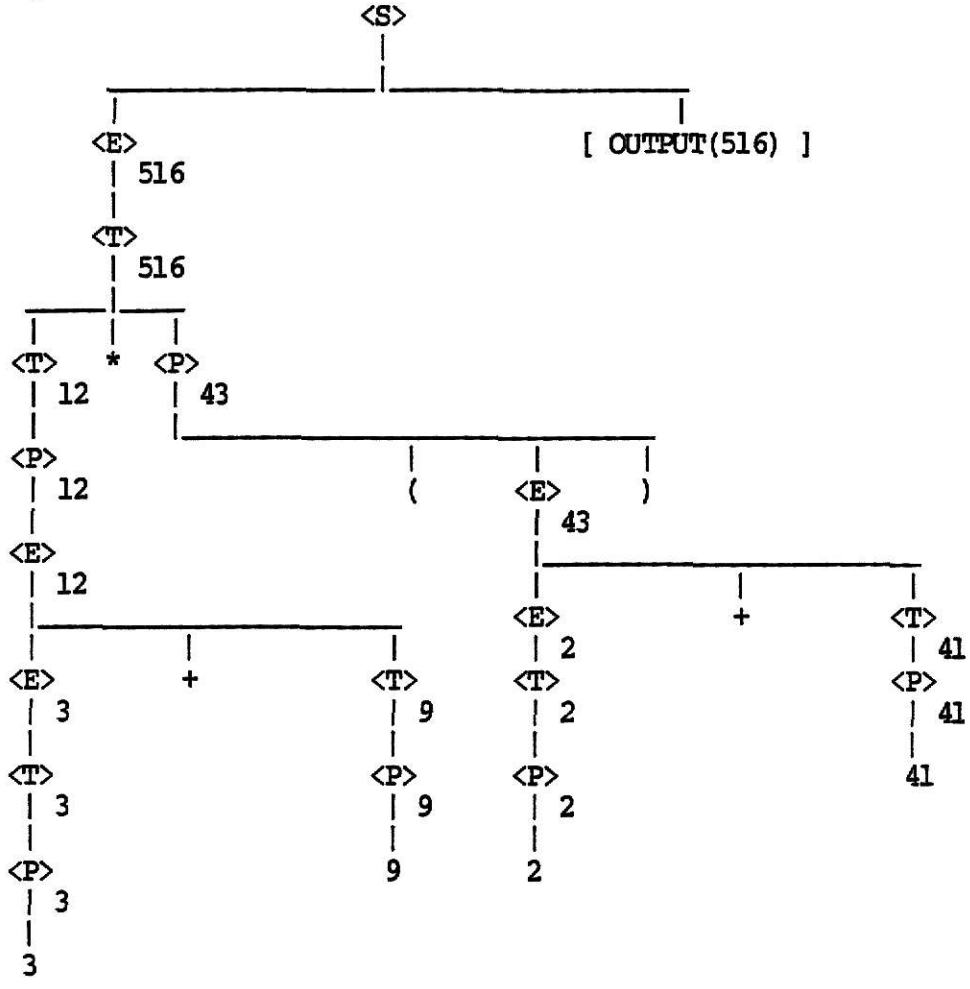


Figure 5: Completed Synthesized Attributes Derivation Tree

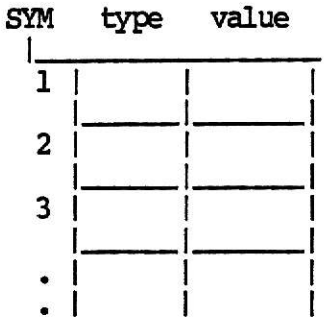
The example should give the reader an idea of how the attributes (or value parts) can be passed up the tree. The 3 and 9 and the 2 and 41 leaf nodes are passed up the tree until they reach the first operator. The four operands are combined into two new value parts and then they continue their ascent up the tree. Finally, the value 516 reaches the top and it is output.

3.2.3 Example 3: Inherited Attributes

In contrast to synthesized attributes (those passed up the tree) are inherited attributes, which are passed down the

derivation tree. That is, the attributes of a right-hand side are inherited from the left-hand side. An appropriate example for this is the construction of a symbol table from

a declaration statement. In this example, the symbol table looks like



and the grammar is

1. <dcl> -> type <v> [p1 <- p; <var-list>
t p t1, t2 <- t; t2
SYM(p1).type <- t1]

2. <var-list> -> , <v> [t1, t2 <- t; <var-list>
t p p1 <- p; t2
SYM(p1).type <- t1]

3. <var-list> -> empty
t

4. <type> -> REAL [t <- REAL] | INTEGER [t <- INTEGER]
t

5. <v> -> l [p <- POS(let)] ("let" is a letter)
p

(the function POS(let) returns position of "let" in the alphabet)

Figure 6: Inherited Attributes Example