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:

[illegible]
Major Professor

LD
2668
.R4
1985
B37
c. 2

A11202 995985

Table of Contents

1. Introduction
2. Overview of Program Development Systems (PDS's)
   2.1 Theory of PDS's
   2.2 Characteristics of PDS's
      2.2.1 Knowledge-Based Approach
      2.2.2 Deductive Characteristic
      2.2.3 High-Level Language Characteristic
      2.2.4 Transformational Approach
      2.2.5 Target Program Domain Oriented vs. General Rule Approach
   2.3 Transformation Rules
   2.4 Survey of Systems
      2.4.1 Structure Editor: Cornell Program Synthesizer
      2.4.2 General Rule Systems
         2.4.2.1 SAFE/TI/GIST
         2.4.2.2 PSI/PMB/PECOS
         2.4.2.3 Dershowitz and Manna
         2.4.2.4 DEDALUS
      2.4.3 Target Program Domain Systems
         2.4.3.1 CPS-G
         2.4.3.2 TRIAD
3. PDS's and Attribute Grammars
   3.1 Definition of an attribute grammar
   3.2 Examples
      3.2.1 Example 1: Translation Grammar
      3.2.2 Example 2: Synthesized Attributes
      3.2.3 Example 3: Inherited Attributes
   3.3 Attribute Grammar use in a PDS
4. A Prototype PDS using an Attribute Grammar
   4.1 PPDS Overview
      4.1.1 Design Approach
      4.2.2 PPDS Description
   4.2 External Grammar
      4.2.1 Token Syntax
      4.2.2 Other Syntax Rules
      4.2.3 Semantic Actions and Symbol Table
         4.2.3.1 Symbol Table
         4.2.3.2 Semantic Action List
      4.2.4 Semantic Rules for External Grammar
      4.2.5 Example
   4.3 Grammar Converter and Internal Representation of Grammar
      4.3.1 Definition Table and Token Table
      4.3.2 Parameter Table, Identifier Table and Constant Table
      4.3.3 Example
   4.4 Interpreter
      4.4.1 Program List
      4.4.2 User Interface
   4.5 Limitations and Extensions
5. Appendix
   5.1 Test Grammar
   5.2 Sample Terminal Session
   5.3 Program Listings
      5.3.1 Converter Listing
      5.3.2 Interpreter Listing

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

Figure 1: Translation Grammar ---------------------- 16

Figure 2: Translation Grammar Derivation Tree ------ 17

Figure 3: Synthesized Attributes Grammar Example ---------------------------------- 19

Figure 4: Synthesized Atrributes Derivation Tree ------------------------------------- 20

Figure 5: Completed Synthesized Attributes Derivation Tree ------------------------- 21

Figure 6: Inherited Attributes Example ------------- 22

Figure 7: Inherited Attributes Derivation Tree ------------------------------------- 23

Figure 8: PPDS Configuration ----------------------- 28

Figure 9: Symbol Table Structure ------------------ 33

Figure 10: External Grammar Example ---------------- 39

Figure 11: Definition Table Structure -------------- 41

Figure 12: Token Table Structure ------------------- 42

Figure 13: Identifier Table Structure -------------- 44

Figure 14: Internal Grammar Example ---------------- 45

Table 1: Fundamental Operations and Structures ---- 27

Table 2: Grammar Syntax Rules ----------------- 31-32

Table 3: Semantic Action List ----------------- 34-36

Table 4: Grammar Semantic Rules --------------- 37-38

Table 5: Limitations and Extensions List ---------- 50

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 adju-stable wrench versus a socket wrench. Obviously, the adjustable wrench is more versatile: able to tighten nuts of many sizes. How-ever, 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. How-ever, 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

```
                                      <E>
                                       |
                                      <T>
               _______________________|_________________________
                |         |                    |                |
               <T>        *                   <P>             [OUTPUT('*')]
                |                              |
               <P>                            <T>
        ________|______                ________|________
         |      |    |                  |              |
         (     <E>   )                  c            [OUTPUT('c')]
 ______________|______________
  |       |        |         |
 <T>      +       <T>      [OUTPUT('+')]
  |                |
 <P>              <P>
  |                |
  |_________       |_________
  |        |       |        |
  a        |       b     [OUTPUT('b')]
           |
      [OUTPUT('a')]
```

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.

example (3+9) * (2+41)

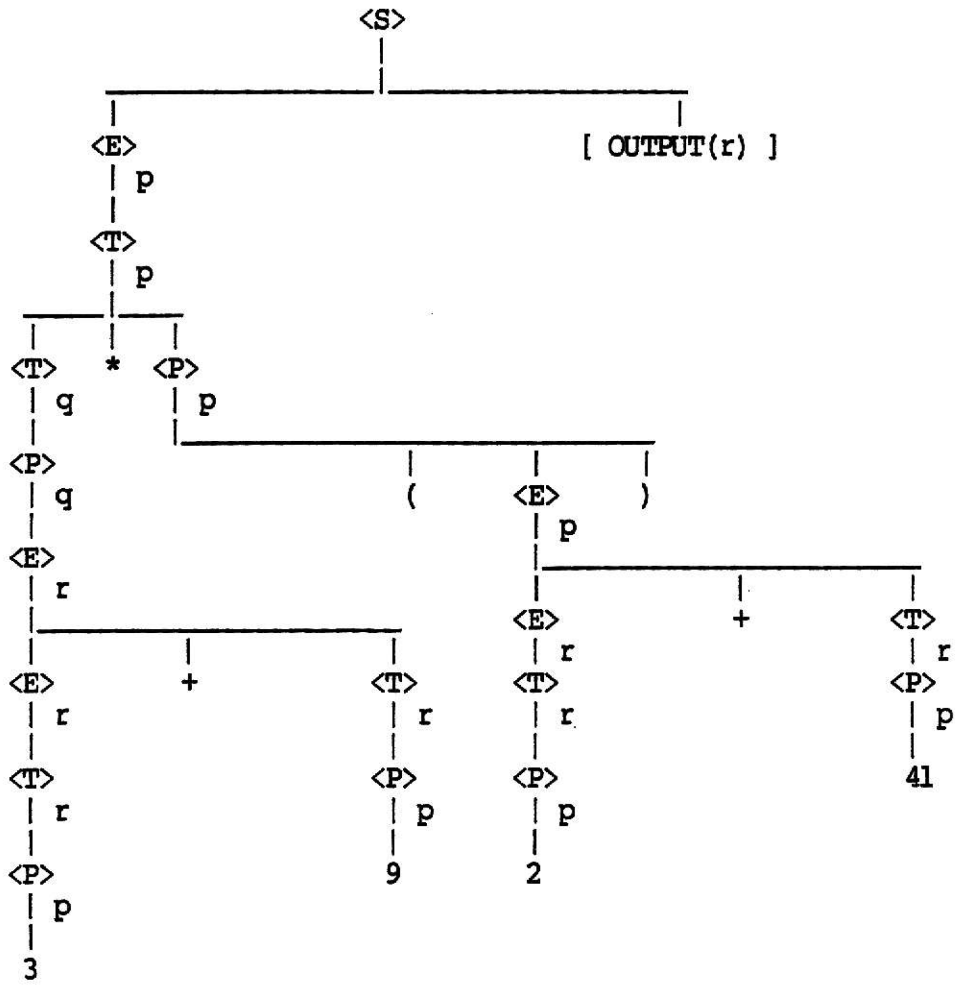


Figure 4: Synthesized Attributes Derivation Tree

The complete tree:

```
                       <S>
                        |
       _________________|____________________
       |                                    |
      <E>                             [ OUTPUT(516) ]
       | 516
       |
      <T>
       | 516
 ______|____
 |     |   |
<T>    *  <P>
 | 12      | 43
 |         |_______________________________
<P>                       |       |      |
 | 12                     (      <E>     )
 |                                | 43
<E>                               |________________________
 | 12                             |            |          |
 |________________________       <E>           +         <T>
 |          |            |        | 2                     | 41
<E>         +           <T>      <T>                     <P>
 | 3                     | 9      | 2                     | 41
 |                       |        |                       |
<T>                     <P>      <P>                     41
 | 3                     | 9      | 2
 |                       |        |
<P>                      9        2
 | 3
 |
 3
```

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

```
       SYM   type   value
        |________________
        1 |      |       |
          |______|_______|
        2 |      |       |
          |______|_______|
        3 |      |       |
          |______|_______|
        . |      |       |
        . |      |       |
```

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> -> 1  [ p <- POS(let) ]       ("let" is a letter)
      p
```

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

Figure 6: Inherited Attributes Example

The derivation tree, complete with semantic actions, of "REAL A,B,C" is

```
                          <dcl>
   _________________________|_______________________________
   |                |              |                       |
 <type>            <v>     [ SYM(1).type <- REAL ]    <var-list>
   |   REAL         | 1                                    | REAL
   |                |                                      |
 REAL               A      ________________________________|__
                          |         |        |                 |
                          ,        <v>  [SYM(2).type <- REAL] <var-list>
                                    | 2                        |REAL
                                    |                          |
                                    B                          |
                 __________________________________________    |
                 |          |          |                       |
                 ,         <v>   [ SYM(3).type <- REAL ]  <var-list>
                            | 3                                |REAL
                            |                                  |
                            C                                empty
```

Figure 7: Inherited Attributes Derivation Tree

The attributes REAL & POS(a) (which is 1) were first passed up the tree to <type> and <v>. REAL was then assigned to the type field of location 1 in the symbol table and then it was passed down the tree being associated with <var-list>. In this subtree, POS(b) (which is 2) was passed up to <v>, REAL was again assigned to the type field in the symbol table, this time to location 2 and passed on down the <var-list> subtree. Similar action was repeated in this subtree until finally the derivation terminated. The reader should be able to visualize how the symbol table would appear at each step in the derivation.

3.3 Attribute Grammar use in PDS's

After looking at three preliminary examples, the real topic of interest can be addressed: how can an attributed grammar be used to generate programs in a PDS? It has long been recognized that many programs contain many of the same fundamental code constructs. In fact, "don't re-invent the wheel" has become a programming cliche. One way this idea can be applied to a PDS is to identify and encode these fundamental constructs into a grammar. Since every case where these constructs will be used is a little bit different, to be truly useful, there must be some way to parameterize the grammar. This can be done through semantic actions. They can be used to prompt the user for input, such as identifier names, array indexes, and the operation desired, to store this information and to recall it when needed in the grammar. The PDS can then produce a complete, correct, compilable program by parsing the grammar.

There are two primary advantages of this approach:

- The programmer will not have to rewrite the fundamental code constructs for each new application. Instead they can be encoded into a grammar and parameterized via semantic actions. Using the PDS, the programmer will then have to simply enter the appropriate parameters.
- By encoding correct syntax into the grammar, the programmer will not have to be concerned with syntactic details (which are tedious and error prone, yet quite critical). The PDS will automatically generate a syntactically correct program.

With these two goals in mind, the challenge becomes to

1) develop a clear, powerful grammar notation in which to encode the fundamental code constructs,
2) develop a set of semantic actions which are complete and easy to use, but also easy to implement,
3) write a program which will parse or interpret this grammar and execute the semantic actions and also conveniently display the program as it is developed,

4) finally, with these tasks done, to build a library of grammars which will generate a large number of fundamental code constructs.

The first three goals were the focus of the the development of the prototype PDS. The fourth was not, although one such grammar was developed for use in the demonstration of PPDS.

4. A Prototype PDS using an Attribute Grammar

4.1 PPDS Overview

4.1.1 Design Approach

The prototype PDS, PPDS, was designed and implemented employing an attribute grammar. It was aimed at a user audience of student programmers in a course on standard algorithms and data structures similar to CMPSC 300 "Algorithmic Processes" at KSU. Although in its implemented state it is just a prototype, it incorporates and illustrates the fundamental principles of a PDS. Furthermore, the task of extending its capabilities to a point of genuine utility would primarily be one of continuing farther in the direction already begun rather than one of heading in a completely new direction. That is, what is mainly needed for the extension is more "brute force" to cover more cases.

The general design methodology used in PPDS was similar to the one used in the TRIAD system: both PDS's use attribute grammars and they both are target program domain oriented. As in the TRIAD system, fundamental code constructs for developing programs in specific target domains were identified and encoded into an attribute grammar. The system generates compilable code by applying the productions in this grammar and through interaction with the user. The grammar is parameterized via the semantic actions in it and so, by accepting input from the user, the program may be tailored to the specific case at hand.

Peterson, in [PE83], observed that beginning programming classes repeatedly write programs using the same collection of operations on the same collection of structures. A PDS which would query a user as to what kind of operations and on what type of structures, and would then automatically generate a correct program would be a quite useful tool in such classes to demonstrate programming style and how different operations are implemented with different structure.

The operations and structures which Peterson identifies are

OPERATIONS	STRUCTURES
Create Structure	Arrays
Examine Structure	1-dimensional,
Assignment	2-dimensional, etc.
Reduction	Linked List
Selection	Trees
plus others	plus others

Table 1: Fundamental Operations and Structures

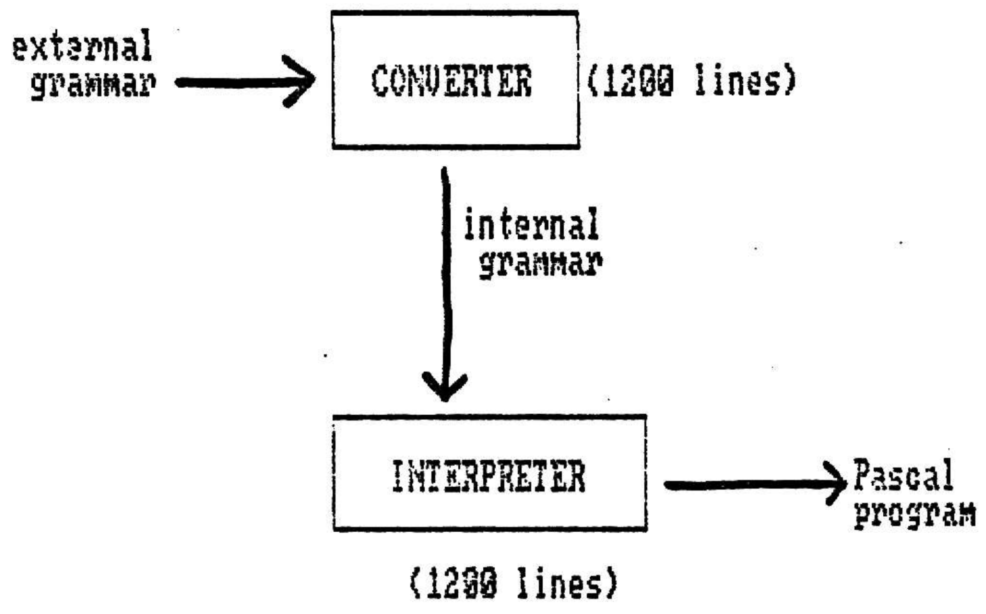
PPDS is not capable of developing programs containing all of the operations and structures in Peterson's list. This list does, however, form the basis of what structures and operations PPDS can handle.

In the survey of systems, the Cornell Program Synthesizer, a structure editor was mentioned. It ensures that a syntactically correct program is written by presenting the user with statement templates and then allowing only syntactically valid information to be entered into the templates. PPDS extends this idea by presenting the user with program or procedure templates and prompting the user for semantically and pragmatically valid input. PPDS then updates the template according to this input.

4.1.2 PPDS Description

The system consists of two components: the converter and the interpreter. (see Figure 8 below) The system was written in Turbo Pascal on an IBM-PC compatible micro-computer. Each component is about 1200 lines of code.

Figure 8: PPDS CONFIGURATION



Briefly, a user must first load a grammar defining the domain of programs to be generated. The grammar is converted to an internal representation by the converter, the first component of the system, and written out to a disk file. To start actual program development, the interpreter, the second component of the PDS, is run. It loads the internal grammar file and presents the user with a series of partially developed program templates that are expanded into more complete ones. The user will be queried to enter information such as identifier names and operations desired. To present the templates, the system traces through the productions in the grammar expanding the templates according to the productions, like a parser. When a production is expanded, its right hand side is grafted into its place in the partially developed program. Semantic actions in the grammar call for input from the user or referencing a symbol table. The symbol table is where the PDS keeps track of all user defined identifiers. The PDS also monitors the values of variables in the grammar. A sample terminal session and the code for PPDS are presented in the appendix.

The two components of PPDS, the external grammar, the semantic actions, the symbol table and the internal grammar, are described below.

4.2 External Grammar

The heart of the PDS is the grammar. It determines how powerful and versatile the system is. The grammar is entered to the system in a standard BNF-like text form: a series of productions with the left-hand sides (LHS's) being expanded into a series of tokens on the right-hand sides (RHS's). It is then converted into an internal linked representation for use by the PDS by the component in the system called the "grammar converter". The external grammar syntax is explained below.

The external grammar is composed of tokens of several different classes. The majority of tokens are simply terminals or non-terminals like one would find in any ordinary grammar. In addition, there are semantic actions which do not correspond to any physical symbols in an input or output stream but rather specify an action to be executed. Also, there are two pre-defined reserved word operators 'begin' and 'end' in the grammar which delimit a scope. These are necessary because there can be overloading of names in the grammar. Static scoping rules apply, but there can be no nested scopes: everything is either global or local one level deep.

For all tokens except semantic actions, the token is surrounded by angle brackets '<' and '>'; semantic actions are enclosed in square brackets '[' and ']'. Tokens are composed of a number of fields, separated by commas. The first field identifies the type of token.

The grammar structure is described in detail below and an example is presented in Figure 10 on page 39.

4.2.1 Token Syntax

The syntax for the metasymbols of the grammar is explained below. Symbols which are underlined are literal strings, the others are pattern placeholders with the following meanings:

integer --	a one-digit unsigned (positive) or negative integer according to Pascal syntax
string --	any sequence of characters except a single quote
identifier --	a identifier according to Pascal syntax, limited to lower case letters, first 15 characters significant

<const, 'string'>
constant token: the value is enclosed in quotes in the second field.

<nt, identifier>
non-terminal: the second field gives the name of the non-terminal.

<inv, identifier>
"invisible" token: token will not be displayed on the screen. They are used as place holders in the partially developed program in case a return to a point in the middle is needed to further expand the program. As with the regular non-terminals, the second field gives the name.

<id, identifier>
identifier token: the second field gives the name of the identifier. Identifiers are most often used as variables to hold information relevant to the program being developed, such as user input. A variable is not explicitly declared. Rather, it is declared implicitly when used and also typed by the context in which it was used. This means the grammar developer must be very careful to use names and types correctly. The two types of variables are integer and string.

<cr, integer>
carriage return in the grammar: the comma and integer field are optional and denotes how many spaces to increased (unsigned integer) or decrease (negative increase) the level of indentation in the next line.

4.2.2 Other Syntax Rules

- The token on the left-hand side of a production is always a non-terminal, so the type field should be left off of these tokens.
- Comments in the grammar are preceded by a double asterisk and terminated by the end of a line.
- a production is terminated by a semi-colon.
- any number of spaces or blank lines can be placed between tokens
- the entire grammar is terminated by an exclamation point
- strings must be 15 characters or less
- 'begin' and 'end' must appear between productions

- any blank spaces that are desired must be included as constants in the grammar

Table 2: Grammar Syntax Rules

4.2.3 Semantic Actions and Symbol Table

In addition to the tokens listed above, there are also semantic actions. They are represented in the grammar enclosed in square brackets '[' and ']', with their parameter list enclosed in parentheses '(' and ')'. Legal types for parameters are the same as for the other grammar metasymbols, as described above.

A list of each semantic action and its function appears below in section 4.2.3. When appearing in the grammar, semantic action names must appear in all upper case letters. Many of the semantic actions operate on the symbol table that is kept by PPDS. So, the structure called SYM_TBL must be explained before dealing with the functions of the semantic actions. The symbol table data structure is explained below.

4.2.3.1 The Symbol Table

In the table are stored the attributes of all identifiers entered by the user during the program development session. The symbol table is accessed by the semantic actions. Various semantic actions call for information to be accessed from the symbol table (eg, SEARCH and REF) or stored into the table (eg, STORE and ENTER). The semantic action list in the section below gives a complete description of how each one interacts with the table.

The table is necessary to store the attributes of identifier in case the attributes are needed later in the program. For example, the index range name is stored in the symbol table entry for an array identifier. Perhaps later in the program the array is used in a FOR loop. The lower and upper bound of the index range of the array would be retrieved from the symbol table and used as the initial and final values on the FOR loop. The format of the symbol table is given below:

```
type

  SYM_TBL_REC is record
                  NAME:  IDENTIFIER
                  CLASS:  IDENTIFIER

                  case  CLASS of
                     'label',
                     'stdtype':  "null"

                     'var':       VAR_TYPE: IDENTIFIER

                     'subrange': LB: integer
                                 UB: integer

                     'array':     INDEX_RANGE_NAME:  IDENTIFIER
                                  ELEM_TYPE:         IDENTIFIER

                     'list':     "if the class is list then the
                                 name field holds the record name"
                                 DATA_FIELD_NAME:  IDENTIFIER
                                 DATA_TYPE:        IDENTIFIER
                                 PTR_FIELD_NAME:   IDENTIFIER
                                 PTR_TYPE:         IDENTIFIER

                     'ptr':      OBJECT_TYPE:      IDENTIFIER
                endcase
              endrecord

TYPE_SYM_TBL is array [1 to ?] of SYM_TBL_REC
```

Figure 9: Symbol Table Structure

This is the "conceptual" symbol table. The actual one implemented has indexes stored instead of names stored in several of the fields. For example, instead of storing the name of the index range of an array, the table stores the index in the symbol table of the index range. The standard types "integer", "real" and "char" are loaded into the first three positions in the symbol table, all with their class set equal to "stdtype". From the declaration, the reader can see that based on the "CLASS" of an identifier, certain other information is stored. For example, if an identifier is the name of an array, the symbol table will store the name of the index range and the name of the element type for the array.

4.2.3.2 Semantic Action List

The list describes each semantic action by telling

- whether a parameter it is an input or an output parameter or both an input and output parameter.
- the type of each parameter, that is, whether it is a string constant, an integer constant or a variable. If it is a variable, the the type of variable is specified.
- its function, including interaction with the user or the symbol table.

Table 3: Semantic Action List

```
1 ID_GET(out ID: string var)
     - prompts user to enter an string value (ie, the name
       of an identifier) to assign to the variable "ID"

     - reads the string user enters
       (no syntax checking is performed presently)

     - ID <— user input
```

```
2 INT_GET( out N: string var)
    - prompts user to enter an integer value  to assign
      to variable  "N"

    - reads the integer user enters
      (the value is read as a string; presently no syntax
       checking is done)

    - N <--  user input

3 ENTER(in INDEX: integer var, in ID_NAME: string var,
       in CLASS: string)

    -  enters a new identifier into symbol table

    -  SYM_TBL[INDEX].NAME  <--  ID_NAME

    -  SYM_TBL[INDEX].CLASS  <--  CLASS


4 STORE(in INDEX: integer var, in FIELD_NAME: string,
       in VALUE: string var)

    - stores information into a specific field of an
      existing entry

    - SYM_TBL[INDEX].FIELD_NAME <-- VALUE


5 REF(out VAR: string var, in INDEX: integer var,
     in FIELD_NAME: string)

    -  references a value in a field in the symbol table

    -   VAR  <---  SYM_TBL[INDEX].FIELD_NAME

6 COMPARE(in VAR: string var, in VALUE: string)

    -  if  VAR  = VALUE
          then  continue with this production
          else  skip to  next production
```

7 CHOOSE(out SELECTION: string var, in CHOICE_COUNT: integer,
in CHOICE_{1} : string ... $\text{CHOICE}_{\text{choice\_count}}$: string)

- this semantic action can have a variable number of parameters. CHOICE_COUNT indicates how many choices there are in the choice list.

- CHOICE_{1} ... $\text{CHOICE}_{\text{choice\_count}}$ is a list of strings which are the choices

- displays CHOICE_{1} through $\text{CHOICE}_{\text{choice\_count}}$ and prompts user to select a choice

- accepts choice entered by user

- SELECTION <— the number of user's choice

8 SEARCH (out INDEX: integer var, in FIELD_NAME: string,
in VALUE: string)

- INDEX <— X such that SYM_TBL[X].FIELD_NAME = VALUE

- SEARCH find the first and only the first entry in the symbol table that meets the conditions. It may be necessary to have a SEARCH that accepts multi-conditions and/or creates a list of all indexes that satisfy the condition(s).

9 ASSIGN (out VAR: integer or string var,
in VALUE: integer or string)

- VAR <— VALUE

- types should match, but no type checking is done

10 COPY (in SOURCE_VAR: integer or string var,
out TARGET_VAR: integer or string var)

- TARGET_VAR <— SOURCE_VAR

- types should match, but no type checking is done

11 INC (in/out VAR: integer var)

- VAR <— VAR + 1

12 DEC (in/out VAR: integer var)

- VAR <— VAR - 1

Table 3: Semantic Action List

If invalid parameters are given in a semantic action, the converter will not detect it. However, when a user tries to run the interperter, it will not work correctly. So, the grammar developer must supply the correct parameters.

4.2.4 Semantic Rules for the External Grammar

Besides the purely syntactic rules about the grammar, there are also semantic rules which must be followed in order to produce a correct, working grammar. A list of these rules follows:

- No "complex type definitions" are allowed. What is meant by this is that a type definition cannot appear inside a type definition. Only type names can appear within type definition. For example, instead of designing a grammar to produce

  ```
  type
    <id_name> = array[<lb>..<ub>] of <elem_type>
  ```

  one should design the grammar to produce:

  ```
  type
     <range_name> = <lb>..<ub>
     <id_name> = array[<range_name>] of <elem_type>
  ```

- No type definitions are allowed in the "var" section, only type names. These two rules are included solely to make the implementation of the converter and interpreter easier, the number of semantic action small and the symbol table simple.

- The grammar can accept only 1 array type and/or list type. This is due to the limitations in the SEARCH semantic action as explained earlier.

- All variables in the grammar are implicitly declared and typed. Great care must be taken in typing variable names so no mistakes are made.

- Alternative productions can only appear if a COMPARE semantic action was included as the first token in the previous production. The alternative must be specified as separate production, but omitting the non-terminal token on the LHS of the alternative productions. Furthermore, an alternative must always follow a production with a COMPARE in it.

Table 4: Grammar Semantic Rules

The person who designs the grammar must be very careful to do so correctly. The dangers with the state variables have already been mentioned. The same care must be taken with semantic actions. If used improperly, a semantic action may access a non-existent, undefined or incorrect entry in the symbol table. These errors would go undetected by the converter, but would cause an error if a user tried to run the interpreter. Essentially, the converter is like a programming language compiler. It can check the syntax, but a successful conversion/compilation hardly guarantees the grammar/program is correct. A small example grammar is included below to help resolve any confusing points.

4.2.5 Example

The example is a portion of the grammar that was used to develop PPDS. It generates the program header and part of the declaration section for a Pascal program. Later in the report, the internal representation of the same grammar is given. The example illustrates the grammar syntax and the use of semantic actions.

```
<pgm> —> <const,'program'> <nt, pgm_id> <const, ';'> <cr,3>
         <nt,dcls>  <cr>
         <nt,body> <cr>;

<pgm_id> —>  [ID_GET(pgm_id)]  [ASSIGN(last, 1)]
             [ENTER(last,pgm_id,'label')] <id,pgm_id>;

<dcls> —> <const,'type'> <cr,3> <nt,types>
                                <inv,more_types> <cr,-3>
            <const,'var'>   <cr>    <nt,vars>
                                  <inv,more_vars>   <cr>
            <nt, procs>   <inv,  more_procs>  <cr>;

<types>  —>  [CHOOSE(structure_type, 2, 'array', 'list')]
             <nt, select_type>;

<select_type> —> [COMPARE(structure_type,'array')]
                         <nt,array_type>;

              —> [COMPARE(structure_type,'list')]
                         <nt,list_type>;

              —> ;


<vars>  —>  <const,' '>;

<procs>  —>  <const,' '>;

<body>  —>  <const,' '>;

<array_type> —> ;

<list_type> —> ;

!
```

Figure 10: External Grammar Example

4.3 Grammar Converter and Internal Representation of Grammar

Before the PDS can process the grammar, the external representation must be converted to an internal form which is much easier to work with. The conversion is performed by a component in the system called, not surprisingly, the grammar converter. It parses the external grammar and converts it to the internal representation: a linked structure in the form of a collection of tables. The representation for the grammar is taken largely from [CO71].

The conversion need only be done when the grammar is first loaded or changed somehow. As long as the grammar remains unchanged, the PDS will operate without having to perform the conversion. The error checking the converter does is minimal. If an error is found in the input grammar, no fix-up is taken: the parser just stops and issues a message telling what the error was and where in the grammar the error was found. The algorithm for the converter is not tremendously complicated; it is a simple parsing algorithm. It has a length of about 1200 lines of code. For specific details about the converter logic, refer to the program listing in the appendix.

The internal structure that the converter generates has five distinct data structures: the definition table, the token table, the semantic action parameter table, the identifier table and string constant table. The structures of the internal grammar are described in detail below and an illustration of the internal representation of the external grammar in section 4.3.3 in presented in Figure 14 on page 45.

4.3.1 Definition Table and Token Table

First, there is a *definition table* with entries consisting of a non-terminal name (essentially the LHS of a production) and an index into the token table corresponding to the first token in the non-terminal's RHS. Also in the definition table is a integer field indicating the scope of the non-terminal (0 for global or 1 for local).

Its structure is shown in the following figure.

```
type  IDENTIFIER  is string[10]

      DEF_REC is record

                       SCOPE     :  integer
                       NAME      :  IDENTIFIER
                       RHS_NDX   :  integer

                endrecord

      DEF_TBL is array[1 to MAX_NUM_NTS] of DEF_REC
```

Figure 11: Definition Table Structure

The token table, which, as stated, is essentially the RHS of a production, is another array of records. Its structure is

```
type TOK_REC is  record
                 |
                 | TOK_TYPE:  string[4]
                 |
                 | case  TOK_TYPE of
                 | |     'cr':    INDENT        : integer
                 | |
                 | |     'inv',
                 | |     'const': NAME          : integer
                 | |
                 | |
                 | |     'id':    ID_TBL_INDEX  : integer
                 | |
                 | |     'nt':    DEF_TBL_NDX   : integer
                 | |
                 | |     'sa':    SA_NAME       :  string[6]
                 | |              PARAM_TBL_NDX:  integer
                 | |
                 | |              case  SA_NAME
                 | |              | 'COMPARE':  ALT: integer
                 | |              |
                 | |              | otherwise: "null"
                 | |              endcase
                 | endcase
                 |
                 | SUCC:  integer
                 |
                 endrecord
```

TOK_TBL is array[1 to MAX_NUM_TOKENS] of TOK_REC

Figure 12: Token Table Structure

The first field is the token type. The types are the same as the ones listed in the discussion of the external grammar, with the addition of 'sa' for semantic actions. While some of the fields are self-explanatory, others need more explanation. The ID_TBL_NDX field is an index into the identifier table (described below) of a particular grammar variable. The field DEF_TBL_NDX points to the entry for the non-terminal in the definition table. The field PARAM_TBL_NDX, for semantic actions, is an index to the start of the parameters for this semantic action into the PARAM_TBL, described below. Furthermore, if the token is a COMPARE semantic action, a field which is a index into TOK_TBL to the first token of an alternative production is included. The field SUCC in the token is an index to the next token in the production. The last token in the production has a zero value for SUCC.

4.3.2 Parameter Table, Identifier Table and Constant Table

Since the semantic actions can have long and variable number parameters, it was decided to put them in their own separate structure, the parameter table, rather than include them as fields in the token table. From examination of the explanation of the semantic actions given earlier, it is evident that all parameters are one of four types: string constant, integer constant, string variable or integer variable. In this implementation, the parameter table is simply an array of integers. For a parameter which is a variable or string constant, the entry in the parameter table is simply an index into another table (the identifier table or the constant table, respectfully--both are described below) where the actual name and/or value of the variable or string is contained. Finally, for integer parameters, the value is stored right in the parameter table. Since the number of parameters is known for each one of the semantic actions (the number is either fixed or can be calculated in the case of CHOOSE), it is not necessary to insert a delimiter to indicate the the end of the parameters for each semantic action. Instead, the PDS will know exactly how many parameters are need for each one.

The other data structures involved in the internal grammar representation are the constant table and the identifier table, where string constants and grammar variables semantic action parameters are stored. The structure for the identifier table is

```
type     ID_REC is record
                         NAME     :   IDENTIFIER
                         VAL_TYPE:    string[1]
                         case VAL_TYPE of
                            'S':  SVAL : string[15]
                            'I':  IVAL :  integer
                         endcase
                      endrecord
```

Figure 13: Identifier Table Structure

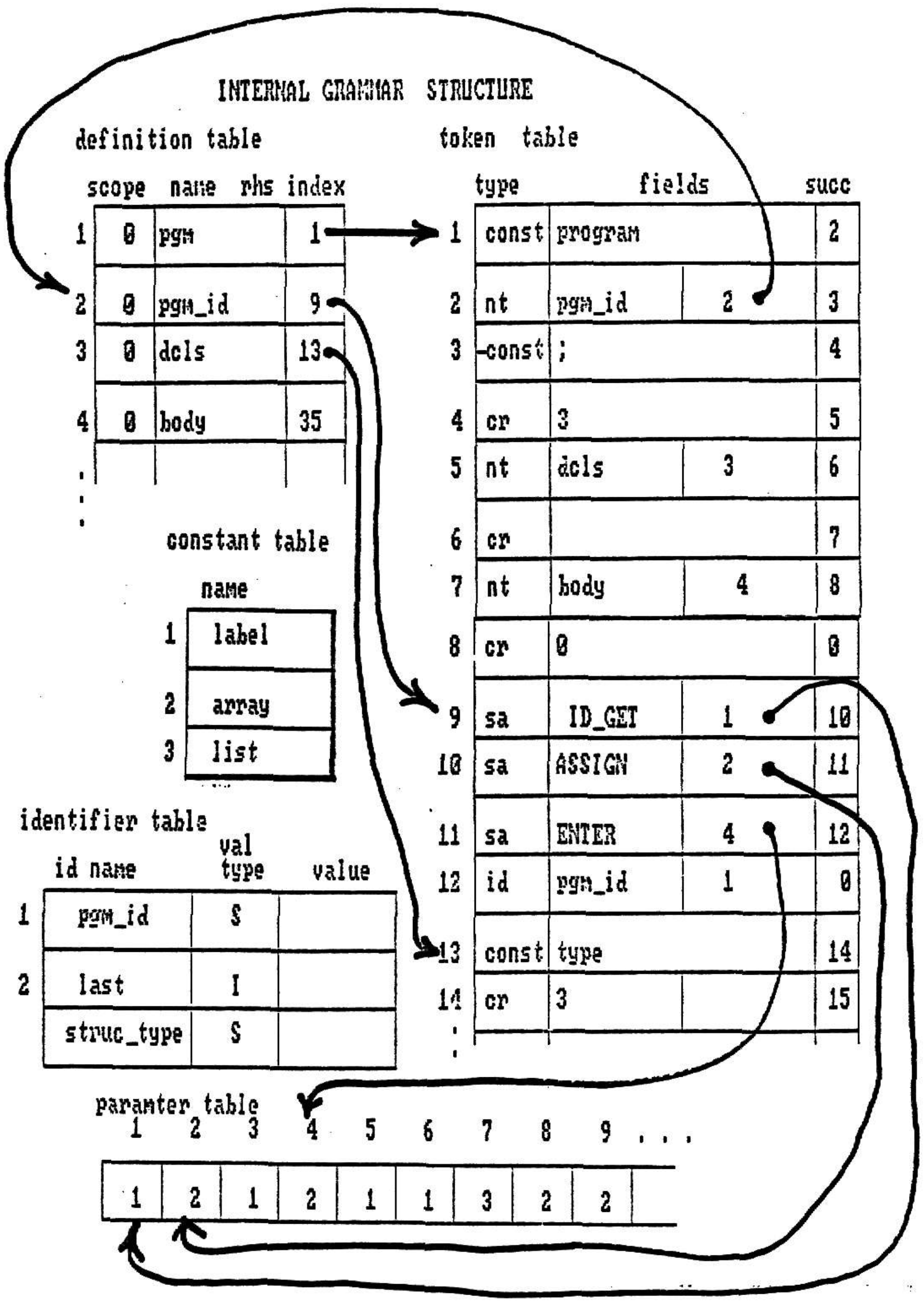
The VAL_TYPE field tells whether the entry is an integer state variable ('I') or string state variable ('S'). Based on the VAL_TYPE, a state variable, another field of the appropriate type is included for the value of the variable. However, in the converter stage, no values are assigned to any variables.

String constant semantic action parameters are stored in a separate table called the constant table. This has a simple structure:

```
CONST_TBL is array[1 to ?] of IDENTIFIER
```

4.3.3 Example

To aid the reader in understanding the internal representation of the grammar an example is included. Figure 14 is a partial internal representation of the external grammar given in Figure 10.



definition table

	scope	name	rhs index
1	0	pgm	1
2	0	pgm_id	9
3	0	dcls	13
4	0	body	35
.			

token table

	type	fields		succ
1	const	program		2
2	nt	pgm_id	2	3
3	const	;		4
4	cr	3		5
5	nt	dcls	3	6
6	cr			7
7	nt	body	4	8
8	cr	0		0
9	sa	ID_GET	1	10
10	sa	ASSIGN	2	11
11	sa	ENTER	4	12
12	id	pgm_id	1	0
13	const	type		14
14	cr	3		15
:				

constant table

	name
1	label
2	array
3	list

identifier table

	id name	val type	value
1	pgm_id	S	
2	last	I	
	struc_type	S	

paramter table

1	2	3	4	5	6	7	8	9	. . .
1	2	1	2	1	1	3	2	2	

Figure 14: Internal Grammar Example

4.4 Interpreter

Once the converter has translated the grammar into the internal form of the five tables, the other main component of the system, the interpreter, can operate. The interpreter is also about 1200 lines of code. The function of the interpreter is to parse through the grammar and generate a complete, compilable program. The basic method used is to build the program by, beginning with the start symbol, replacing non-terminals with their right-hand sides. When all productions have been completed, the program is complete. The logic behind the process is explained in more detail below.

The first task of the interpreter is to load the 5 tables produced by the converter: the definition table, the token table, the parameter table, constant table and the identifier table. The interpreter uses the tables to parse the grammar. Also, it creates two more data structures of its own: the program list and the symbol table. The symbol table was described earlier; the program list is is discussed later.

When the 5 tables have been loaded, the parsing can begin. The interpreter begins with the start symbol, which is found as the first entry into the definition table. The start symbol is copied from the table and becomes the first node in the program list, which is explained in the next section.

4.4.1 Program List

The program list is a doubly linked list, each node being a token. At any time, the contents of the list is the current state of the program that is being developed. After the list is initialized to the start symbol, as the interpreter parses the grammar, the list grows and is refined until finally, when the parsing is complete, the program list is a complete program.

The list grows and is refined by the interpreter performing a sequential node by node transversal of the list. Based on the type of token in the current node, different action is taken:

```
IF   the current token is a non-terminal,
THEN the RHS of the non-terminal is copied from the token
     table and grafted into the program list to replace
     the non-terminal node. The first token in the RHS
     then becomes the current node and the transversal
     continues.

IF   the current token is an identifier,
THEN the value of the identifier is retrieved from the
     identifier table and is inserted into the program
     list as a constant token to replace the identifier
     node.

IF   the current token is a semantic action,
THEN a routine is called to perform the semantic action.
     Depending on which semantic action it is, it may
     prompt the user for some input or somehow access the
     symbol table.

IF   the current token is a carriage return

THEN The current level of indentation is increased/decreased
     by the number of spaces specified in the token

IF   the current token is a constant or invisible,
THEN nothing is done. The interpreter merely moves ahead to
     the next node.
```

When the interpreter finally reaches the end of the program list, if the grammar was designed correctly, all non-terminals, identifiers and semantic actions should be removed. The remaining tokens should all be carriage returns, constants or invisible (which merely serve as place holders). This final version of the list is the complete, compilable program which was desired. The sample terminal session in the appendix should give the reader of how the program list develops.

4.4.2 User Interface

Besides the logic and structures, another important feature of the interpreter is the user interface. Basically, what the user will see on the CRT screen is the program list. It appears as a series of partially developed program templates which will become more developed and refined as the interpreter progresses through the program list. Periodically, the user will be prompted to enter information (such as identifier names and range bounds) or select a choice from a menu list. In this way, the user guides the system to develop a program that satisfies the specific requirements of the case at hand.

The initial template is simply the start symbol. Each time the interpreter replaces a non-terminal in the program list with its RHS or replaces an identifier with its value, the screen is redrawn with the updated version of the program list. Not all nodes in the list are displayed: only non-terminals, identifiers, constants and carriage returns. Semantic actions and invisible tokens are not displayed on the screen.

The display routine is written so that the current token is always displayed in lower brightness in the middle of the screen. It is kept in the middle by traversing the program list in reverse starting with the current token and counting the number of carriage returns encountered. When the number reaches 12 (half the total number of lines displayable on the screen), the spot is marked as the point at which to begin the display. If there are less than 12 carriage returns before the current token, enough blank lines are written to ensure the current token is centered in the screen.

A desirable feature in the user interface would be the ability to scroll up and down through the displayed form of the program list. Although it is desirable, the present version of the interpreter does not allow it. Once a certain point in the program list has been past, there is no way to return to that point.

Although not part of the display routine, the execution of certain semantic actions is part of the user interface. When the interpreter encounters a "CHOOSE", "ID_GET", or "INT_GET" semantic action, the user is prompted for some type of input. The prompts occur at the bottom of the screen. For exactly how the user is prompted, refer to the section in the interpreter program listing dealing with the semantic actions. For exactly what the user is prompted for and what is done with the user input, refer to section 4.2.3 on semantic actions. One important note about the user interface is that no verification of the input is performed. Whatever the user enters is accepted.

4.5 Limitations and Extensions

Because PPDS was meant to merely be a demonstration of the idea of using an attribute grammar in a PDS and was not meant to be commercial quality, it contains many limitations and lacks many desirable features. Many of these have been pointed out in previous sections. In this section, all of the limitations and desirable extensions are collected and presented together.

- no verification of user input in "CHOOSE", "ID_GET" and "INT_GET"
- implicit declaration and typing of variables in the grammar
- "SEARCH" only finds the first occurance of a given value in the symbol table. This means the grammar can develop programs with at most 1 structure of each type.
- a return to an earlier point in the grammar marked by a "inv" token is not implemented
- the user can not scroll up and down through the displayed version of the program list
- there is very limited syntactical error checking and absolutely no semantic error checking of the grammar done by the converter or the interpreter
- there may be useful or even essential semantic actions which are not included
- there is no way for a grammar developer to create his own semantic actions
- there is no way to chain together different grammars

Table 4: Limitations and Extensions List

BIBLIOGRAPHY

Primary References

1. [BA79] Barstow, D. R. "An Experiment in Knowledged-Based Automatic Programming", Aritficial Intelligence, vol 12 (August 1979), pp. 73-119.

2. [CO71] Cohen and Gottlieb. "Table Driven Parsing" Computing Surveys, vol 2., number 1 (March, 1970), pp. 65 - 81. 1971.

3. [LE76] Lewis, P. M. II; Rosenkrantz, D. J.; Stearns, R. E.; Compiler Design Theory; Addison-Wesley Publishing Co. (1976). pp 181-212.

4. [PA83] Partsch, H. and Stienbruggen, R. "Program Transformation Systems", Computing Surveys, vol. 15, no. 3 (September 1983), pp. 99-236.

5. [PE83] Peterson, Gerald. "Using Generalized Programs in the Teaching of Computer Science", ACM SIGSCE Bulletin, vol. 15, number 1 (February, 1983), pp. 187 - 191.

6. [RA81] Ramanathan, J.; "Modeling of Problem Domains for Driving Program Development Systems", ACM Proceedings on Programming Languages (1981).

Additional References

1. [BA76] Balzer, R., Goldman, N. and Wile, D. "On the Transformational Implementation Approach to Programming", Proceedings of 2nd International Conference on Software Engineering (San Francisco, 10/13-15, 1976). IEEE Press, New York, pp. 337-344.

2. [DE77] Dershowitz, N. and Manna, Z. "The Evolution of Programs: Automatic Program Modification" IEEE Transactions on Software Engineering, SE-3, number 6 (1977) pp. 377-385.

3. [GR82] Green, C., et. al. "Research on Knowledge-Based Programming and Algorithm Design", 1981 Rep. Kes. U. 81.2, Kestrel Institute, Palo Alto, CA.

4. [MA78] Manna, Z. and Waldinger, R. "DEDALUS--The DEDuctive ALgorithm Ur-Synthesizer", Proceedings of National Computer Conference(Anaheim, CA June 5-8, 1978), vol 47, pp. 683-690.

5. [RE84] Reps, Thomas and Tietlebaum, Tim. "The Synthesizer Generator", ACM SIGPLAN Notices, May, 1984, pp. 42-48.

6. [TE80] Teitelbaum, Tim and Reps, Thomas. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Department of Computer Science, Cornell University, Ithica, NY, 1980.

7. [WI77] Wile, D., Balzer, R., and Goldman, N. "Automated Derivation of Program Control Structures from Natural Language Program Descriptions" in Proceedings of Symposium on Artificial Intelligence and Programming Languages (Rochester, NY, 8/15-17, 1977). SIGPLAN Notices (ACM), vol. 12, no. 8 (Aug 77) pp. 77-84.

5. Appendix

NOTE: The following sections:

- 5.1 Test Grammar
- 5.2 Sample Terminal Session
- 5.3 Program Listings
  - 5.3.1 Converter Listing
  - 5.3.2 Interpreter Listing

are not included in this copy because of their length. These sections can be obtained through the Department of Computer Science, KSU.

A PROGRAM DEVELOPMENT SYSTEM USING AN ATTRIBUTE GRAMMAR

by

KIRK BARRETT

B.S., Kansas State University, 1982

AN ABSTRACT OF 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

ABSTRACT

This report discusses the use of automated program development systems (PDS's) to produce high-level language programs. A prototype PDS, designed and implemented to develop standard algorithm Pascal programs is also included. A working definition for a PDS is an environment in which a user can develop correct, compilable programs without entering code character by character.

PDS are based on the principle that knowledge about programming can somehow be expressed in some machinable form to apply the knowledge to generate programs. A fundamental difficulty is that programming knowledge is highly intuitive and therefore not easily machinable.

To be truly useful, a PDS should apply knowledge to all three areas of programming: syntax, semantics and pragmatics. Such a PDS will free the programmer from the tedious and error prone, yet critical, job of producing a syntactically correct program. It will automatically ensure that syntactic constructs are combined in ways which are semantically valid, and rapidly produce correct programs which solve real, useful problems without complete, full-cycle, unassisted development by the programmer.

Several PDS's which have appeared in literature are discussed in the report and are grouped into two semi-distinct categories: general rule and target program domain systems. The former can develop programs independently of the domain of the program, while the latter can only develop programs within a certain domain.

Some PDS's are based on the observation about programming that programs are composed of a relatively small number of fundamental structures and operations. In the prototype PDS, PPDS, knowledge about fundamental structures and operations is encoded into an attribute grammar. An attribute grammar is a language whose productions contain semantic actions. Here, the productions generate code syntax and the semantic actions are symbol table operations and prompts for user input. By interpreting this grammar, the PDS produces correct, compilable Pascal programs. The code for PPDS, an example grammar and a sample terminal session are included in the appendix.

1430-60
CD-53