# An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases

Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC 29634
U.S.A.
malloy@cs.clemson.edu

James F. Power
Department of Computer Science
National University of Ireland
Maynooth, Co. Kildare
Ireland
James.Power@may.ie

*Abstract*— **We present a structured reformulation of the seminal algorithm for automatic generation of test cases for a context-free grammar. Our reformulation simplifies the algorithm in several ways. First, we provide a structured reformulation so that it is obvious where to proceed at each step. Second, we partition the intricate third phase into five functions, so that the discussion and comprehension of this phase can be modularized. Our implementation of the algorithm provides information about the grammatic, syntactic and semantic correctness of the generated test cases for two important languages in use today: C and C++.**

**The results of our study of C and C++ highlight a lacuna latent in the research to date. In particular, if one or more of the automatically generated test cases is syntactically or semantically incorrect, then the confidence of structural "*coverage*" may be compromised for the particular grammar-based tool under test. Our ongoing work focuses on a solution to this problem.**

*Keywords*— **Structural-based testing, implementation-based testing, black-box testing, white-box testing, context-free grammar, parsing, re-engineering.**

## I. Introduction

With the burgeoning popularity of XML, and other grammar-based tools such as C++, Java, OCL and domain specific languages, the demand for robustness and correct functioning has intensified the importance of testing. One form of testing applied to tools and other software is specification-based testing, where the testing criteria is coverage of the requirements or specification of the software[5], [8], [18], [24]. Specification-based testing attempts to validate the functionality of the software without consideration of the code itself. One drawback of specification-based testing is that some parts of the code are likely to remain unexercised, lowering confidence in the robustness of the software. Thus, most developers favor a testing strategy that exploits both specification-based and implementation-based testing of the code[7], [14], [27].

The seminal paper addressing the issue of automatic generation of test cases to test grammars and grammar-based tools is the work of Purdom for generating sentences from a context-free grammar[23]. The goal of Purdom's algorithm is to use each production in the grammar at least once and to rapidly generate a minimal set of sentences that are short. Several researchers have either based their test case generation on Purdom's work, or an extension of the work including references [3], [6], [16] and [20]. Recent research has addressed the problem of test case generation for attribute grammars[12], [13].

However, the expression of the Purdom algorithm makes comprehension difficult and explication of the algorithm cannot be based on consideration of the algorithm alone. For example, there are places in the algorithm where it is difficult to determine what is to be done after completion of a given step. To illustrate, in step 11 of phase 3 of the algorithm it is obvious that one should not continue to step 12 of the algorithm, yet no further direction is given for an alternative next step. However, by consideration of the sample grammar offered in explanation, it is possible to disambiguate much of the algorithm.

Nevertheless, even using the statement of the algorithm in tandem with the sample grammar, one cannot unequivocally determine the meaning of the algorithm. For example, in the sample grammar, the algorithm reaches completion when the grammar non-terminals move from a state described as "unsure", to a state described as "finished". However, construction of a finite state machine describ-

ing these states of the algorithm reveals that there is no transition from the "unsure" state to the "finished" state. Thus, any use of the seminal Purdom algorithm must include disambiguation and interpretation of the algorithm.

In this paper, we present a structured reformulation of the seminal algorithm for automatic generation of test cases for a context-free grammar[23]. We provide a detailed overview of the critical third phase of the algorithm together with a trace of a sample grammar that threads our paper. Our reformulation simplifies the algorithm in several ways. First, we provide a structured reformulation so that it is obvious where to proceed at each step. Second, we partition the intricate third phase into five functions, so that the discussion and comprehension of this phase can be modularized. Our implementation of the algorithm provides information about the grammatical, syntactic and semantic correctness of the generated test cases for two important languages in use today: C and C++.

The results of our case study with C and C++ highlight a lacuna latent in the research to date. In particular, if one or more of the automatically generated test cases is syntactically or semantically incorrect, then the confidence of structural "*coverage*" may be compromised for the particular grammar-based tool under test. Our ongoing work focuses on a solution to this problem.

In the next section, we provide background about grammars, parsers and testing, including a distinction between sentences and test cases. In Section III we present our structured reformulation of Purdom's algorithm and in Section IV we present a case study using our implementation of the algorithm. In Section V we overview research about test case generation from grammars and in Section VI we describe the problem of defining adequacy criteria for testing a compiler front-end. In VII we draw conclusions.

## II. Background

In this section we define terminology associated with context-free grammars and describe a mapping from programs to grammars. We describe the symmetry of parsing and generating sentences and conclude the section with an overview of software testing. A general description of languages, grammars and parsing can be found in references [2] and [19]. An overview of testing can be found in references [5] and [18].

### A. Terminology

Given a set of words (known as a lexicon), a *language* is a set of valid sequences of these words. A *grammar* defines a language; any language can be defined by a number of different grammars. When describing formal languages such as programming languages, we typically use a grammar to describe the *syntax* of that language; other aspects, such as the semantics of the language typically cannot be described by context-free grammars. Extended Backus-Naur Form (EBNF) is a commonly-used notational enrichment of context-free grammars which does not enhance their descriptional power.

Formally a grammar is a four-tuple $(N, T, S, P)$ where $N$ and $T$ are disjoint sets of symbols known as non-terminals and terminals respectively, $S$ is a distinguished element of $N$ known as the start symbol, and $P$ is a relation between elements of $N$ and the union and concatenation of symbols from $(N \cup T)$, known as the production rules. A grammar defines a language by specifying valid sequences of derivation steps which produce sequences of terminals to form the *sentences* of the language.

The procedure of using a grammar to derive a sentence in its language is as follows. We begin with the start symbol $S$ and apply the production rules, interpreted as left-right rewriting rules, in some sequence until only non-terminals remain. This process defines a tree whose root is the start symbol, whose nodes are non-terminals and whose leaves are terminals. The children of any node in the tree correspond precisely to those symbols on the right-hand-side of a production rule. This tree is known as a *parse tree*; the process by which it is produced is known as *parsing*.

Figure 1 illustrates "little", a sample grammar that we use as illustration of terminology and the phases of Purdom's algorithm. little consists of three production rules, numbered (0), (1), and (2) on the left side of the figure, non-terminal and start symbol $S$, non-terminal $E$, and terminals + and $ID$. The parse tree, illustrated on the right side of Figure 1, contains the start symbol as root and the leaves represent the sentence $ID + ID$.

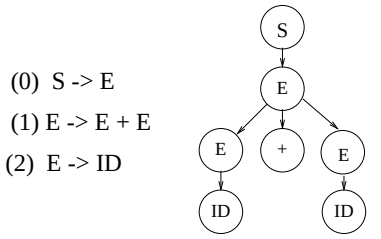


Fig. 1. *Sample grammar.* This figure illustrates "little", a sample grammar that threads our paper. We use little to illustrate the phases of Purdom's algorithm.

### *B. Generating Test Cases*

Generating test cases to provide coverage of a grammar-based tool is a two-phase process. In the first phase sentences are generated consisting of non-terminals and in the second phase the non-terminals are transformed to produce a string in the language. The first phase is analogous to parsing and proceeds as follows: begin with the start symbol $S$ and apply the production rules until only non-terminals remain. Purdom's algorithm addresses the problem of sentence generation; the problem of transforming the non-terminals in the sentence to produce a test case has been addressed elsewhere[3], [6], [20].

A *test case* is a pair, *(input, expected result)*, in which *input* is a description of an input to the software under test and *expected result* is a description of the output that the software should exhibit for the associated input[18]. In our paper, the input is the grammar under test; the expected output is the meaning of the test case. The *correctness* of a test case is a measure of the accuracy of the model of the software. The degree of accuracy is judged with respect to a standard that is assumed to be infallible, referred to as an *"oracle"*. The oracle is often a human expert whose domain knowledge is sufficient to be used as a standard.

### *C. Testing*

Software testing is the evaluation of the work products created during a software development effort[18]. Some developers view the testing effort as the most important effort of the development process; there are those who are "test infected", so that they cannot begin to code until they have written test cases[9]. The test cases determine when coding is complete; when the developer cannot produce test cases that break the system, the system is completely done[4]

Given the importance of the testing process, it is surprising that there are so many approaches to testing, with no testing technique recognized as current best practice. There are advocates for each approach, including those that favor specification-based testing, those that favor implementation-based testing and those that favor a hybrid approach. In *specification-based testing*, the tester generates test cases based on the software specification, without considering the code itself[5], [8], [18], [24]. In *implementation-based testing*, the tester generates test cases based on the code; these approaches are usually based on knowledge about the flow of control or the flow of data through the program[15], [17], [26]. Some developers favor a hybrid approach that combines the advantages of both specification-based testing and implementation-based testing[7], [14], [27]. However, no approach to testing has gained overall acceptance in the software community.

## III. Purdom's Algorithm: reformulation & interpretation

In this section we present our interpretation of Purdom's algorithm for generating short sentences for a grammar, with each production in the grammar used at least once. In our interpretation, we have made every effort to remain faithful to the original algorithm. In many cases, we use the same variable names for the important data structures used in the algorithm. Moreover, our reformulation, though structured, parallels the steps as expressed in the seminal paper[23].

Input to the algorithm is a context-free grammar with a unique *unused* start symbol: the start symbol does not appear on the right hand side of a production rule. To generate sentences, the algorithm must be able to choose the production that will produce the shortest terminal string and it must use each production at least once. To achieve these goals the algorithm uses various data structures in three phases. In the sections that follow, we discuss each of these phases beginning with the data structures and then proceeding with an intuition of how the particular phase accomplishes its goal. We describe, in detail, the algorithm for the particular phase under consideration.

SLEN

| | |
|---|---|
| + | 1 |
| ID | 1 |
| S | 3 |
| E | 2 |

RLEN

| | |
|---|---|
| 0 | 3 |
| 1 | 6 |
| 2 | 2 |

SHORT

| | |
|---|---|
| S | 0 |
| E | 2 |

Fig. 2. *Phase one data structures.* This figure illustrates the final values in SLEN, RLEN and SHORT for little, the sample grammar that threads our paper.

```
algorithm    Shortest terminal string
variable     SLEN : terminals & non-terminals
output       RLEN : productions
output       SHORT: non-terminals
( 1)  void init() {
( 2)    for ( each terminal t ) { SLEN[t] = 1; }
( 3)    for ( each non-terminal n ) {
( 4)        SLEN[n] = max_int;
( 5)        SHORT[n] = -1;
( 6)    }
( 7)    for ( each production p ) {
( 8)        RLEN[p] = max_int;
( 9)    }
(10)  }

(11)  void PhaseOne() {
(12)    bool change = true;
(13)    while ( change ) {
(14)        change = false; {
(15)        for ( each production p ) {
(16)            int sum = 1; bool too_big = false;
(17)            for ( each element e of RHS[p] ) {
(18)                if ( SLEN[e] == max_int ) {
(19)                        too_big = true; break;
(20)                }
(21)                else sum += SLEN[e];
(22)            }
(23)            if ( !too_big && sum < RLEN[p] )
(24)                RLEN[p] = sum;
(25)                if ( sum < SLEN[LHS[p]] )
(26)                  SHORT[LHS[p]] = p;
(27)                  SLEN[LHS[p]] = sum;
(28)                  change = true;
(29)                }
(30)            }
(31)        } // for
(32)    } // while
(33)  } // PhaseTwo
```

Fig. 3. *Phase one of Purdom's algorithm.*

### *A. Phase One: Shortest Terminal String*

The first phase of Purdom's algorithm uses three data structures, SLEN, RLEN and SHORT; SLEN stores values for terminals and non-terminals, RLEN stores values for each production and SHORT stores values for each non-terminal. The deliverable of this first phase, essential for the third phase is SHORT; however, SLEN and RLEN, used in phase two, facilitate the computation of SHORT.

At the termination of the first phase, SLEN contains the number of steps in the shortest derivation of each symbol in the data structure; for each terminal SLEN contains the "size" of the terminal and for each non-terminal SLEN contains the number of steps in the shortest derivation of a sentence starting with that non-terminal. At termination, RLEN stores the length of the shortest string that can be derived for each production and SHORT stores, for each non-terminal, the corresponding production number to use to derive the shortest string starting with the respective non-terminal.

Figure 2 illustrates the values in these data structures at the end of phase one for the sample grammar little, presented in Section II (see Figure 1). On the left, Figure 2 illustrates the final values for SLEN, where each terminal is size 1, and the non-terminals $S$ and $E$ are size 3 and 2, representing the shortest derivations of $S$ and $E$. The shortest derivation of $S$ can be viewed as a parse tree with $S$ as the root, $E$ as interior node and $ID$ as a child of $E$; clearly such a tree is size 3. The middle of Figure 2 illustrates RLEN, the number of steps in a derivation starting with the symbol on the left side of the production. For example, RLEN[1] is 6, which is the number of steps required to derive a parse tree starting with non-terminal $E$ and using the right side of production 1; this is seen more easily by counting the number of nodes in the parse tree of Figure 1 rooted at $E$, which represents the sentence derived starting with the left side of production 1. The right of Figure 2 illustrates SHORT, the data structure containing the number of the production to use to derive the shortest sentence starting with this non-terminal. For example, the shortest string that can be derived with non-terminal $E$ must begin with the left side of production 2.

As an intuition for computing the value of SHORT, needed in the third phase of the algorithm, consider that given SLEN and RLEN, SHORT can be deduced in isolation; that is, the grammar is not needed to deduce SHORT. The algorithm for finding the elements of SHORT proceeds as follows: find a non-terminal

DLEN

| | |
|---|---|
| S | 3 |
| E | 3 |

PREV

| | |
|---|---|
| S | Null |
| E | 0 |

Fig. 4. *Phase two data structures.* This figure illustrates the final values in DLEN and PREV for little, the sample grammar that threads our paper.

in SLEN, find the production in RLEN whose left hand side matches the non-terminal in SLEN, and then place an entry in SHORT for that non-terminal and that production number.

Figure 3 contains a structured reformulation of this first phase of Purdom's algorithm. In the figure, the lines of code numbered 1 through 10 illustrate the initialization phase of the algorithm, and the lines numbered 11 through 33 illustrate the computation of the values for SLEN, RLEN and SHORT. The initial values of SLEN are 1 for terminals, line 2, and max_int for non-terminals, line 4. Initial values for SHORT are $-1$, line 5, and initial values for RLEN are max_int, line 8.

The algorithm to compute the values in Figure 2 is illustrated in Figure 3 as function PhaseOne(), lines 11 to 33. The algorithm is iterative: the **while** loop permits iteration over the productions using the outer **for** loop, lines 15 to 31, until there are no more changes to SLEN (line 25). The variable sum is initialized to 1 at the start of the inner **for** loop. sum accumulates the value in SLEN for each element on the right hand side of the current production; if any value of SLEN is max_int, then this production is skipped. If the production is not skipped, then the value of sum is compared with the value of RLEN for this production; if sum is less than the current value of RLEN for this production, line 23, then the value of RLEN is updated at line 24. If sum is also less than the current value of SLEN for the non-terminal on the left hand side of this production, line 25, then the value of SHORT for the left hand side of this production is updated at line 26; that is, the current production will lead to a shorter derivation than the one currently stored in SHORT. The value of SLEN is updated similarly on line 27, and change is updated, line 28, to reflect that the **while** loop should iterate over the productions again (until there are no more changes).

The important fact about phase one of the algorithm is that SHORT must contain, for each non-terminal, the production number that will lead to the shortest derivation for this production.

```
algorithm   Shortest derivation
input       SLEN : terminals & non-terminals
input       RLEN : productions
variable    DLEN : non-terminals
output      PREV : non-terminals

( 1)  void init() {
( 2)    for ( each non-terminal n ) {
( 3)        DLEN[n] = max_int;
( 4)        PREV[n] = -1;
( 5)    }
( 6)  }

( 7)  void PhaseTwo() {
( 8)    bool change = true;
( 9)    while ( change ) {
(10)        change = false; {
(11)        for ( each production p ) {
(12)            if ( RLEN[p] == max_int ) continue;
(13)            if ( DLEN[LHS[p]] == max_int ) continue;
(14)            if ( SLEN[LHS[p]] == max_int ) continue;
(15)            sum = DLEN[LHS[p]] + RLEN[p]
(16)                          - SLEN[LHS[p]];
(17)            for (each non-terminal n on RHS[p]) {
(18)                if ( sum < DLEN[n] ) {
(19)                  change = true;
(20)                  DLEN[n] = sum;
(21)                  PREV[n] = p;
(22)                }
(23)            } // for
(24)        } // for
(25)    } // while
(26)  } // PhaseTwo
```

Fig. 5. *Phase two of Purdom's algorithm.*

### *B. Phase Two: Shortest Derivation Algorithm*

The second phase of Purdom's algorithm uses SLEN and RLEN, computed in phase one, together with a local data structure, DLEN, to produce PREV; PREV is the deliverable of this second phase and is essential to the third phase of the algorithm. There are entries in DLEN and PREV for each non-terminal. At termination of this second phase, for each symbol $s$, DLEN will contain the length of the shortest derivation that uses $s$. PREV will contain the number of the production to use to introduce $s$ into the shortest derivation; thus, since the start symbol is not introduced by any symbol, it's value is *Null*.

Figure 4 illustrates the values in DLEN and PREV at termination of the second phase of the algorithm.

ONCE

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| S | R | 0/R | R | U | U | F |
| E | R | 1 | R | R | F | F |

MARK

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | f | t | t | t | t | t |
| 1 | f | t | t | t | t | t |
| 2 | f | f | f | t | t | t |

ONST

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| S | 0 | 1/0 | 0 | 0 | 0 | 0 |
| E | 0 | 1 | 2 | 1 | 0 | 0 |

Fig. 6. *Phase three data structures.* This figure illustrates the values in ONCE, MARK and ONST for little, the sample grammar that threads our paper. The first column of boxes lists the non-terminal, production numbers and non-terminals respectively for the structures; subsequent columns list values at the beginning of phase three, and then at the end of the do_sentence loop in Figure 8 for iterations 1 through 5 of the loop.

The values for DLEN, on the left of the figure, are 3 for the non-terminals: the length of the shortest derivation of $S$ and $E$ is 3. The values for PREV, on the right of the figure, are $Null$ for the start symbol, and 0 for E. Thus, to introduce $E$ into a shortest derivation, production 0 should be used; note that the left hand side of production 0 is the start symbol.

Figure 5 contains a structured reformulation of this second phase of the algorithm. In the figure, the lines of code numbered 1 through 6 illustrate the initialization phase of the algorithm, and the lines numbered 7 through 26 illustrate the computation of the values for DLEN and PREV. The initial values for DLEN are max_int, and for PREV are $-1$, as shown on lines 3 and 4.

The algorithm to compute DLEN and PREV is illustrated in Figure 5 as function PhaseTwo(), lines 7 to 26. As with phase one, the algorithm is iterative: the **while** loop permits iteration over the productions using the outer **for** loop, lines 11 to 24, until there are no more changes to DLEN (line 18).

### *C. Phase Three: Sentence Generation Algorithm*

The final phase of Purdom's algorithm is the most involved of the three phases. We have made an attempt to simplify the algorithm in several ways. First, we provide a structured reformulation so that it is obvious how to proceed at each step. Second, we partition the algorithm into five functions, illustrated in Figures 7 and 8, so that the discussion and comprehension can be modularized. We begin this section by presenting the important data structures, and then we provide detailed discussion of the algorithm.

```
algorithm   Sentence Generation Algorithm
input       PREV : non-terminals
input       SHORT: non-terminals
variable    MARK : productions
variable    ONST : non-terminals
variable    ONCE : non-terminals
( 1)  void init() {
( 2)    for ( each non-terminal n ) {
( 3)        ONCE[n] = Ready;  ONST[n] = 0
( 5)    }
( 6)    for ( each production p ) MARK[p] = false;
( 9)  }
(10)  int short(int nt) {
(11)    int prod_no = SHORT[nt];
(12)    MARK[prod_no] = true;
(13)    if ( ONCE[nt] != Finished ) ONCE[nt] = Ready;
(14)    return prod_no;
(15)  }
(16)  void load_ONCE() {
(17)    for ( each production p ) {
(18)        if ( MARK[p] == false ) {
(19)            ONCE[LHS[p]] = p;  MARK[p] = true;
(21)        }
(22)    }
(23)  }
(24)  void process_STACK(int & nt, int prod_no,
(24)                     void bool & do_sentence) {
(25)    ONST[nt]--;
(26)    for ( each element e of RHS[prod_no] ) {
(27)        STACK.push( e );
(28)        if ( e is non-terminal ) ONST[e]++;
(29)    }
(29)    bool done = false;
(30)    while ( !done ) {
(31)        if ( STACK.empty() ) {
(32)            do_sentence = false;   break;
(33)        }
(34)        else {
(35)            nt = STACK.top();  STACK.pop();
(37)            if ( is_terminal(nt) ) print(nt);
(38)            else done = true;
(39)        }
(40)    }
(41)  }
```

Fig. 7. *Auxiliary functions, init, short, load_ONCE, and process_STACK for phase three of Purdom's algorithm.*

### C.1 The Data Structures

The data structures for this third phase are illustrated in Figure 6: ONCE, containing an entry for each non-terminal, at the top of the figure; MARK, containing an entry for each production, in the middle of the figure; and, ONST, containing an entry for each non-terminal at the bottom of the figure. The values in ONCE can range over the integers and our productions are numbered starting at 0. However, Once may also contain values of Null, Ready, Unsure and Finished; we assign integer values to these constants: Null is $-1$, Ready is $-2$, and so forth. Thus, the values in ONCE will range from $-4$ to $n-1$ for $n$ productions in the grammar.

The values in the first column of each structure in Figure 6 represent an index into the structure; subsequent columns are the values stored in the structure. The numbers above the structure indicate the iteration through the algorithm. For example, for ONCE, the values for the $0^{th}$ iteration are $R$ for both $S$ and $E$; this indicates that the initial values for $S$ and $E$ are $R$. The values for the $1^{st}$ iteration of ONCE are $0/R$ and 1 for $S$ and $E$ respectively; this indicates that at the end of the first iteration of the algorithm, ONCE[$S$] was 0 and then $R$, ONCE[$E$] was 1. Similarly, at the end of the $2^{nd}$ iteration, ONCE[$S$] and ONCE[$E$] were both $R$.

### C.2 The Auxiliary Functions

Figure 7 illustrates the four auxiliary functions that facilitate computation in the third phase of the algorithm. The functions are init, short, load_ONCE, and process_STACK, Function init initializes the data structures, the values in ONCE to Ready, the values in ONST to 0 and the values in MARK to false; this initialization is accomplished on lines 1 through 9 of Figure 7. Note that these initial values correspond to the values in the $0^{th}$ columns for each structure in Figure 6.

Function short returns the production number that will lead to the shortest derivation for the input parameter nt, a non-terminal, line 14 of Figure 7. Before returning, short also assigns the value **true** to this production number, line 12. Finally, short examines the $nt^{th}$ location of ONCE and, if it's not marked Finished, short marks it Ready, that is, even though the algorithm was not able to assign a production number in the usual way, this production

```
( 1)  void PhaseThree() {
( 2)    bool done = false;  int prod_no = Null;
( 3)    while ( !done ) {
( 4)      if ( ONCE[Start] == Finished ) break;
( 5)      ONST[Start] = 1; nt = Start; do_sentence = true;
( 6)      while ( do_sentence ) {
( 7)        once_nt = ONCE[nt];
( 8)        if ( nt == Start && once_nt == Finished) {
(10)          done = true;  break;
(11)         }
(12)        elsif (once_nt == Finished) prod_no = short(nt);
(13)        elsif (once_nt >= 0) {
(14)          prod_no = once_nt;  ONCE[nt] = Ready;
(15)         }
(16)        else {
(17)          load_ONCE();
(18)          for ( each non-terminal I ) {
(19)            if ( I != Start && ONCE[I] >= 0 ) {
(20)              J = I;  K = PREV[J];
(21)              while ( K >= 0 ) {
(22)                J = LHS[K];
(23)                if ( ONCE[J] >= 0 ) break;
(24)                else {
(25)                  if ( ONST[I] == 0 ) {
(26)                    ONCE[J] = K;  MARK[K] = true;
(27)                   }
(28)                  else ONCE[J] = Unsure;
(29)                 }
(30)                K = PREV[J];
(31)               } //while
(32)             } // if
(33)           } // for
(34)          for ( each non-terminal n ) {
(35)            if (ONCE[n] == Ready) ONCE[n]=Finished;
(36)           }
(37)          if (nt==Start && ONCE[nt]< 0
(38)              && ONST[Start] == 0) break;
(39)          elsif ( ONCE[nt] < 0 ) prod_no = short(nt);
(40)          elsif ( ONCE[nt] >= 0 ) {
(41)            prod_no = ONCE[nt]; ONCE[nt] = Ready;
(42)          }
(43)        } // else
(44)          process_STACK;
(45)       } // while (do_sentence)
(46)     } // while (!done)
(47)   } // PhaseThree()
```

Fig. 8. *Phase three of Purdom's algorithm.*

may still be useful in a derivation. This last step is illustrated on line 13 of Figure 7. Line 13 was not included in the original statement of Purdom's algorithm; it is part of our interpretation.

The last function in Figure 7 is process_STACK, which manages the parse stack used in phase 3. The function begins by decrementing ONST for the current non-terminal, line 25. We note that this decrementation may be out-of-phase with the insertion of the non-terminal onto the stack; that is, the non-

terminal was likely pushed onto the stack on line 27 in a previous iteration of the phase 3 algorithm, and then was possibly popped at line 31. The **while** loop, lines 30 to 40, is our interpretation of correct management of the parse stack, including the processing of all terminals and non-terminals and checking for completion of the sentence, line 32, and possibly the algorithm, line 38.

C.3 Sentence Generation

Sentence generation is choreographed by function PhaseThree, illustrated in Figure 8. The outer **while** loop, extending for most of the function from lines 3 to 46, repeatedly iterates through the data structures until all productions are used and all sentences generated. The inner **while** loop from line 6 to line 45 generates a single sentence.

The inner **while** loop in function PhaseThree has three important parts. In part one, lines 8 to 15, the algorithm attempts to determine if it is finished generating sentences. In part two, lines 17 to 43, production numbers are installed into ONCE to create a path from the start symbol. This path will consist of a production number associated with each non-terminal; either this is a production that has not been used, i.e. MARK was false, or the production that will lead to the shortest derivation for this non-terminal. This assignment to ONCE is accomplished either at line 17 through the function load_ONCE, or line 26, or line 39 where ONCE is assigned the number leading to the shortest derivation.

## IV. Case Study

In this section, we present a case study to demonstate our interpretation of Purdom's algorithm. We developed a test suite of five grammars, including little, grammars from references [12] and [23], the ANSI C grammar, and the GNU C++ grammar. After presenting the test suite, we present the results of using these grammars as input to our interpretation of the algorithm. Finally, we discuss "coverage", as it relates to the grammatic, syntactic and semantic correctness of the generated sentences.

For our implementation, we used the GNU C++ compiler, *gcc* version 2.95.2, running on a 500 MHz Pentium II Dell Optiplex, using RedHat Linux version 6.1. For the data structures discussed in Section III, we used standard library maps, with terminals and non-terminals represented as strings[1]. Sentence generation was efficient: for the GNU C++ grammar, 5.77 seconds were required to generate and print the 81 sentences.

### *A. The Test Suite*

The test suite that we used in our study is illustrated in Figure 9, where the first column lists the name of the test case, the second column lists the number of terminals in the grammar, the third column lists the number of non-terminals, the fourth column lists the number of production rules and the last column lists the average size of the right hand side of the rules of the grammar. This average size includes both terminals and non-terminals.

The first row of the table lists statistics for little and the second row lists statistics for Purdom, the grammar used by Purdom in his original presentation of the algorithm[23]. The third row lists statistics for Harm, the grammar used in reference[12], the fourth row lists statistics for the ANSI C grammar and the last row lists statistics for the GNU C++ grammar.

For the test suite, the smallest grammar is little, with only 2 terminals, 2 non-terminals and 3 rules. The largest grammar is clearly GNU C++, with 110 terminals, 232 non-terminals and 824 rules. The first three test cases, little, Purdom and Harm are academic grammars; the last two test cases, ANSI C and GNU C++, specify languages currently in use in both industry and academia.

### *B. The Generated Sentences*

The results of our study are summarized in Figure 10, where the test cases are listed in the first column and the results are listed in the last four columns. We include the second column, listing the number of rules for each grammar, for comparison of grammar size with the number of generated sentences. The interesting columns in Figure 10 are the third column listing the average size of each generated sentence, the fourth column listing the number of sentences generated, the fifth column listing the number of syntactically correct sentences and the sixth column listing the number of semantically correct sentences.

The average sentence size for ANSI C is larger than for GNU C++, with 33.3 and 21.0 listed in the last two rows of column three. This anomaly is due to the fact that the C grammar is defined by the ANSI

| Test Case | Structure of the test case | | | |
|---|---|---|---|---|
| | Terminals | Non-terminals | No. of rules | Avg size of RHS |
| Little | 2 | 2 | 3 | 1.66 |
| Purdom | 5 | 4 | 7 | 1.85 |
| Harm | 9 | 6 | 11 | 2.18 |
| ANSI C | 83 | 64 | 213 | 2.27 |
| GNU C++ | 110 | 232 | 824 | 2.33 |

Fig. 9. **Test Suite.** This table lists the grammars that we used as test cases for the implementation of our interpretation of Purdom's Algorithm. The column on the left of the table lists the grammar, and the four columns on the right list statistics about the structure of each grammar.

| Test Case | Sentences generated by the algorithm | | | | |
|---|---|---|---|---|---|
| | No. of rules | No. of sentences | Avg size of sentence | No. syntactically correct | No. semantically correct |
| Little | 2 | 1 | 3.0 | N/A | N/A |
| Purdom | 7 | 2 | 7.0 | N/A | N/A |
| Harm | 11 | 3 | 8.6 | N/A | N/A |
| ANSI C | 213 | 11 | 33.6 | 2 | 2 |
| GNU C++ | 824 | 81 | 21.0 | 7 | 5 |

Fig. 10. **Results of the case study.** This figure illustrates the results of our study using five grammars, including C and C++. The column on the left lists the test cases and the three columns on the right list the results of the study. We include the second column describing the number of rules in each test case grammar as a means of comparing the results of each test.

standard committee and likely written to be more readable than implementable. The GNU grammar is written to be acceptable to the Bison compiler generator and has more rules than the ISO C++ grammar[21], which is likely to lead to smaller size sentences.

All of our generated sentences were lexically and grammatically correct. *Lexical correctness* means that the generated symbols are terminals in the grammar; *grammatical correctness* means that the sentences can be derived by repeated application of the rules of the grammar. However, distinguishing between syntax and semantics is not always easy, even in the area of formal methods: "the dividing line between the two areas is not fixed"[25, page 2]. In compiler technology, this line is blurred further because, for a given language, syntax and semantic correctness is determined at different stages depending on the particular compiler implementation.

To evaluate our test cases, we used the GNU *gcc* C and C++ compilers, version 2.95.2, as an *oracle*[1] to judge the syntactic and semantic correctness for the last two test cases in Figure 10: ANSI C and GNU C++. We did not have an oracle for the little, Purdom and Harm test cases. In particular, a sentence is *syntactically correct* if *gcc* considers it correct, using only the grammar. A sentence is *semantically correct* if *gcc* considers it correct, using information gathered from the grammar and additional structures, such as a symbol table[22]. We provide examples below.

For the little, Purdom and Harm test cases, 1, 2 and 3 sentences were generated respectively. The statistics for syntactic and semantic correctness for these test cases are shown as $N/A$, *not applicable*, in columns four and five of Figure 10. For ANSI C, 11 sentences were generated with 2 sentences were both syntactically and semantically correct. For example, *gcc* considered the sentence *"char ;"* to be correct, although it did provide a semantic warning that the sentence contained a *"useless"* keyword. We list this test case as both syntactically and semantically correct.

For GNU C++, 81 sentences were generated with 7 sentences syntactically correct and 5 sentences semantically correct, as illustrated in the last row of Figure 10. The sentences were manually translated into test cases and an effort to generate meaningful test cases was extended for all of the sentences. For example, one generated sentence:

[1] Please see Section II-B for a discussion of oracles.

*"ASM_KEYWORD '(' STRING ')' ';'*,
was translated into a test case as:
*asm (".align 4");*
*gcc* considered this test case to be syntactically and semantically correct. A second interesting example is the generated sentence:
*"USING NAMESPACE IDENTIFIER SEMI"*,
which we translated into a test case as
*using namespace X;*
This test case was considered syntactically correct but semantically incorrect since *"namespace X is undeclared"*.

## V. Related Work

In this section, we overview research about automatically generating sentences to test grammar-based tools. We begin by describing works that extended Purdom's algorithm and conclude the section with a recent application of sentence generation to attribute grammars. Our overview of the research proceeds in chronological order so that each work reviewed appeared in the literature before the next.

Much of the literature describes the generation of both context-free and context-sensitive test cases. A *context-free* test case is a transformed sentence that is syntactically correct. A *context-sensitive* test case requires inclusion of context-sensitive information into the generative process, such as generating a declaration to match a use of a variable, to facilitate production of semantically correct test cases.

### *A. Minimal & Maximal Strategy*

Reference [6] extends Purdom's algorithm to include versions that modify SHORT to return either the non-terminal that will lead to the shortest derivation or the non-terminal that will lead to the longest derivation. Thus, they provide both a *minimal* and a *maximal* strategy for generating sentences. A second extension permits the user to specify that an iterative construct of the grammar may be multiply used, forcing deeply nested structures that might saturate the compiler stacks or tables and test for overflow.

The approach addresses the issue of sentence correctness, permitting the user to extend the grammar by inserting or deleting terminals or non-terminals or to augment sentence generation by including "context-sensitive information"[6]. The user-defined actions might supply a declaration to correspond to the usage of a variable, or to initialize a variable used in computation. The actions can be incorporated into any of the multi-pass phases of generation, permitting a combinatorial product of the sentences that might be generated in isolation. They describe levels of correctness, including *lexical*, *syntactic*, *compile-time*, *run-time*, and *logical* correctness. Finally, the need for systematically generating incorrect sentences is described.

Our ongoing work includes the modification of MARK to store integers rather than boolean values to control the generation of nested structures. We are also developing criteria to describe the adequacy of testing grammar-based tools. We note that Purdom did not consider the issue of sentence correctness, adequacy criteria or the generation of sentences using combinations of grammar rules[23].

### *B. Parametric Grammars*

Reference [3] extends Purdom's algorithm in an effort to provide (1) a large or "inexhaustible" supply of compilable programs, and (2) a controlled method for generating incorrect programs. The technique exploits a *parametric grammar* to describe both syntactic and semantic aspects of the programming language. The parametric grammar is the input to a generator that, together with user-supplied information, drives test case generation. Finally, test cases are "decoded" and printed[3, p. 345].

*Parametric grammar*s are augmented context-free grammars that accept *parameters* to facilitate **generation** of context-sensitive languages rather than **recognition**. The parameters describe language constructs such as identifiers or type information that can be used to incorporate context sensitive information into the process. The generative algorithm is essentially Purdom where function process_STACK, Figure 7 on page 6, is augmented to permit processing of the parameters to the grammar. The technique was used to generate sentences to test four Pascal compilers, including generation of both correct and incorrect programs.

### *C. Pseudo-random Generation*

Reference [20] generates both context-free and context-sensitive sentences, as well as invalid sentences for PT, a subset of Pascal. The generating algorithm is constructed systematically from a set

of production rules that map the syntax into a sequence of statements. This statement sequence is similar to a recursive descent parser except the goal is not to recognize a sentence, but rather to generate one. The "recursive descent" generation algorithm use a pseudo-random number to choose the type of statement, variable or declaration to construct.

The sentence generator has a context-sensitive mode that facilitates generation of correct sentences. When the declarative part of a program block is generated, the information is maintained in a symbol table, creating a contextual environment within which statements are generated. Declarations are removed from "scope" upon exiting the scope. Thus, the approach is limited to PT programs.

### *D. Attribute Grammars*

The work in references [12] and [13] address the problem of test case generation for context-free and attribute grammars, with particular tuning for attribute grammars. They introduce the notion of approximation coverage applicable in two dimensions: syntax and semantics. An equational system to describe the two aspects of coverage is developed. In the case of an attribute grammar for example, the syntactic dimension corresponds to coverage of the underlying context-free grammar and coverage of the semantic dimension corresponds to coverage of the attributes, conditions and computations specified by the attributes.

*Approximation coverage* includes a "layered" definition of coverage for non-terminals. For example, consider production $n- > w$, for $n$ a non-terminal and $w$ one of the alternatives of $n$. This alternative is said to be *covered* by a test suite if all grammar symbols in $w$ are covered; this notion of coverage is in contrast to the notion in Purdom that simply enforces the rule $n- > w$ to be used once by some sentence in the test suite. The work includes recursive unfoldings of the grammar in its sentence generation to produce nested structures. Since in general, full coverage is not feasible, opportunities to relax the coverage are discussed. The notion of non-feasibility of grammar coverage due to certain semantic constraints is comparable to the infeasibility path problem in software testing[28].

The authors present a sentence generation algorithm using decorated derivation trees. Only correct derivation trees are generated starting from an elementary tree that is then completed. The generated test cases are small and redundancy is reduced. Termination of the algorithm cannot, in general, be guaranteed. However, generating test cases from derivation trees is an interesting alternative to the recursive descent approach of reference [20] or the vectored (or mapped) approach of Purdom[23]. The authors are extending their work to apply approximation coverage to a Pascal-like language[13].

## VI. **Adequacy Criteria**

Specification of the criteria to determine what constitutes an "adequate test" is one of the most important aspects of software testing[10]. One criterion for adequacy is *statement coverage*, which requires that every statement in the program is executed at least once. Stronger criteria have been developed such as *branch*, *path* and *data flow* coverage[28]. However, a definition of adequacy in terms of compiler testing has not been developed.

Purdom's algorithm generates sentences so that each rule in the grammar is executed at least once; this *rule coverage* is similar to statement coverage for software testing. The approximation coverage of reference [13] is stronger than rule coverage since approximation coverage requires that, for rule $n- > w$, all grammar symbols in $w$ are covered. Further research to develop a hierarchy of coverages would facilitate proper choice by compiler developers.

However, a more important issue is the effect of grammar-based test case generation on the testing of the **underlying code** for the front-end. For example, how do the test cases generated from the grammar specification effect testing of the code to insert identifiers into a symbol table. To provide adequate coverage of a compiler front-end, test cases generated from a grammar specification must be augmented with test cases for the underlying code. The grammar-based test cases may be essential for testing the semantic actions inserted into the front-end specification. If the grammar-based test cases are incorrect, it is possible that some of the underlying code may never be reached, since the triggering of grammar rules is a *gateway* to the underlying code. Our ongoing work attempts to address these issues.

## VII. Conclusions and Future Work

We have presented a structured reformulation of the seminal algorithm for generating test cases using a context-free grammar. Our reformulation simplifies the algorithm in several ways. First, our structured reformulation makes it obvious how to proceed at each step. Second, our partitioning of the intricate third phase modularizes the discussion and comprehension of the algorithm. We distinguish between *lexical*, *grammatic*, *syntactic* and *semantic* correctness of the sentences generated by the algorithm. Our study includes information about the correctness of the generated test cases for two important languages in use today: C and C++.

Our ongoing work includes an investigation of techniques to generate sentences using combinations of grammar rules as well as the iterative use of recursive rules. We are extending our generation to the ISO C++ and Java grammars[1], [11]. However, the most important focus of our work addresses the adequacy of grammar-based testing, particularly as it relates to testing the underlying code of the compiler front-end.

## VIII. Acknowledgement

The authors extend their appreciation to Steve Stevenson and Murali Sitaraman, together with the members of the Clemson formal methods seminar, who offered input and suggestions about syntactic and semantic correctness.

## References

[1] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. American National Standards Institute, first edition, September 1998.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] F. Bazzichi and I. Spadafora. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering*, SE-8(4):343–353, July 1982.

[4] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[5] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[6] A. Celentano, S. Reghizzi, P. Della Vigna, and C. Ghezzi. Compiler testing using a sentence generator. *Software – Practice and Experience*, 10:897–913, 1980.

[7] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, July 1998.

[8] R. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[9] E. Gamma and K. Beck. Test infected: Programmers love writing tests. Using JUnit to automatically generate test cases, http://members.pingnet.ch/gamma/junit.htm 2001.

[10] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 3, June 1975.

[11] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[12] J. Harm and R. Lämmel. Testing attribute grammars. *Proceedings of the Third Workshop on Attribute Grammars and their Applications (WAGA'2000)*, pages 79–98, July 2000.

[13] J. Harm and R. Lämmel. Two-dimensional approximation coverage. page TBA, TBA 2001.

[14] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. *Proceedings of the 14th International Conference on Software Engineering*, 1992.

[15] M. J. Harrold and G. Rothermel. Peforming data flow testing on classes. *SIGSOFT*, 1994.

[16] W. Homer and R. Schooler. Independent testing of compiler phases using a test case generator. *Software – Practice and Experience*, 19(1):53–62, January 1989.

[17] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen, and J. Gao. Object state testing and fault analysis for reliable software systems. *IEEE 7th Int'l Symp. Software Reliability Eng.*, 1996.

[18] J. D. McGregor and D. A. Sykes. *A Practical Guide To Testing Object-Oriented Software*. Addison-Wesley, 2001.

[19] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan-Kaufman, 1997.

[20] V. Murali and R. K. Shyamasundar. A sentence generator for a compiler for PT, a pascal subset. *Software – Practice and Experience*, 13:857–869, 1983.

[21] J. F. Power and B. A. Malloy. Metric-based analysis of context-free grammars. In *Proceedings of the 8th International Workshop on Program Comprehension*, Limerick, Ireland, June 2000.

[22] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in ISO C++. In *Technology of Object-Oriented Languages and Systems, TOOLS 2000*, pages 57–68, Sydney, Australia, November 2001.

[23] P. Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, April 1972.

[24] M. Roper. *Software Testing*. McGraw-Hill, 1994.

[25] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc, 1986.

[26] A. L. Souter and L. L. Pollock. OMEN: A strategy for testing object-oriented software. *Proceedings of the International Symposium on Software Testing and Analysis*, August 2000.

[27] P. Thevenod-Fosse and H. Waeselynck. Towards a statistical approach to testing object-oriented programs. *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, June 1997.

[28] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12(12):1128–1138, December 1986.