TESTING OF GRAMMARS FOR TOP-DOWN PARSERS

TESTING OF GRAMMARS FOR TOP-DOWN PARSERS

By
ASMA M PARACHA

MS (Computer Eng.)

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Applied Science

McMaster University

MASTER OF APPLIED SCIENCES (2008)
(Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: Testing Grammars For Top-down Parsers

AUTHOR: Asma M. Paracha, M.S. (Sir Syed University of Eng. and Tech.)

SUPERVISOR: Professor Frantisek Franek

NUMBER OF PAGES: ix, 88

Abstract

During the past decades, the complexity of compilers has grown much and so has the importance of testing them. Compiler is essentially a software tool, and hence its testing should fulfill all the software testing criteria. Testing is the process of finding errors in an application by executing it. It is one of the essential and most time consuming phases of software development. Hence a lot of effort is directed to fully automate this process, making it more reliable, repeatable, less time consuming, less boring, and less expensive. Test cases for compiler should be generated so that they cover all possible valid and invalid input conditions. One of the major problems in generating test cases is the completeness of coverage, and the potentially unfeasible size of the generated test data. The test data for compilers should ideally cover all the syntax and semantic rules of the language in all possible combinations and in all possible contexts. When generating test cases for a compiler, the grammar act as the foundation as it defines the language for which the compiler is being built. In this research we addressed the issue of automatic generation of test data for parsers by implementing Purdom's algorithm in Java and C++ and generating test data for MACS compiler.

Acknowledgments

I cannot praise enough my supervisor Dr. Frantisek F. Franek who has been a source of inspiration and leading light during the course of my thesis. It would have been impossible to bring the project to the stage where it now stands, without his guidance and moral support. The time spent with him has been an experience, and I hope that it would help me in the challenging life lying ahead.

Finally, without the love, support, and prayers of my parents, husband, and my loving kids, I could not have put my best in this project and I am truly grateful to them.

Contents

List of Figures

List of Tables

Chapter 01
Introduction

During the past decades, the complexity of compilers has grown much and so has the importance of testing them. Compiler is essentially a software tool and hence its testing should fulfill all the software testing criteria. The test data for compilers should ideally cover all the syntax and semantic rules of the language in all possible combinations and in all possible contexts. One of the major problems in generating test cases is to ensure the completeness of coverage and the potentially unfeasible size of the generated test data. If upon executing a test case, the output matches the excepted one (including the error messages generated), then the compiler passed the test. On the other hand, if the generated output and/or errors if applicable do not match, the compiler has errors and should be corrected.

Compilation is the process of transformation of the source program written in a source (input) language to a program in an target (output) language. Typically, (since the advent of Algol 68 language), the syntax of a source language is specified by means of a formal context-free grammar. The grammar than is the main input for the test-case-generation process. A grammar not only defines a language, it also provides a basis for deriving elements of that language, thus in software engineering terms, the grammar is considered both a specification and a program.

To generate a sentence in top-down manner in a language using a grammar, we begin with the start symbol ***S*** of the grammar and apply production rules interpreted as left-right rewriting rules in some sequence until we are only left with a sentential form containing only terminal symbols. This process is known as syntax analysis, or more commonly as parsing. This process may

generate a tree whose root is the start symbol ***S***, whose internal nodes are labelled by non-terminals and whose leaves (terminal nodes) are labelled by terminals (often referred to as tokens). The children of an internal node ***A*** in the tree correspond precisely to the symbols on the RHS (right-hand side) of a production rule with ***A*** as its LHS (left-hand side) symbol. Such a tree is known as a parse tree. Note that often more concise form of parse trees are used, so-called syntax trees.

Testing a grammar for errors is difficult. A grammar should be tested to verify that it defines the intended language and that it is complete in the sense that every non-terminal has some terminal derivation. Detecting errors in the grammar at an early stage is very important as the construction of the compiler depends on it.

It is important to remark on the relationship of a programming language and its grammar or grammars: no context-free grammar can define a programming language fully as it cannot capture the context–sensitive aspects, or, even if it could, it would make the grammar prohibitively big and unwieldy. Thus most of context-sensitive aspects and some other aspects (e.g. a requirement that a variable be defined/declared before it is used) are left to semantic analysis phase of compilation and is not dealt with at the syntax level.

Our thesis focuses on checking of LL(1) grammars for MACS and generating test data for MACS compilers' parsers. MACS is an object oriented language created by Prof. Franek for his forthcoming book on compilers; its syntax is similar to C++ and Java, but somehow simpler. There are two versions of MACS compiler, one programmed in C++ and the other programmed in Java. The C++ MACS compiler has a bison-generated bottom-up MACS parser based

on a LALR grammar. The Java MACS compiler has a JavaCC-generated top-down MACS parser based on an LL(1) grammar.

In this project, we have implemented the Purdom's algorithm to (a) test for completeness the various LL(1) grammars as generated from MACS LALR grammar, and (b) to generate short MACS programs as test data for the JavaCC-generated MACS top-down parser.

While researching the history of Purdom's algorithm, we came across a number of difficulties, as there is very little literature available for Purdom's algorithm, though the algorithm is referred to and cited quite often. The algorithm was designed by Purdom [29] in 1972 and only a very high-level logic in imperative style description was given. In the original description it is not clear when the algorithm is to stop generating sentences or how the two main routines, the parsing and the referee routine, communicate. Some work was done later on by Malloy and Power [21,22] and they described the Purdom's algorithm in a more structured separated into three phases. Our implementation is based on their reformulation of the algorithm into the phases. However, their reformulation has also some problems and discrepancies in the third and the most important phase, which generates the sentences. Our main contribution is in implementing the third phase of the algorithm in a different way and successfully generating test data (i.e. short MACS programs).

The thesis is structured in the following way. In Chapter 2 we present an overview of context-free grammars and formal languages and introduce the BNF notations for grammars. In Chapter 3 we present a brief overview of parsing techniques (both top-down and bottom-up) with focusing more on top-down predictive parsing and LL(1) grammars. Chapter 4 deals with the software engineering perspective of compiler validation, while Chapter 5 discusses

compiler (validation) testing, including a survey on test-generation methods. Chapter 6 includes the main contribution, the high-level description of our implementation of Purdom's algorithm. The last chapter presents a conclusion. In the appendices, the final LL(1) MACS grammar is given.

The code of our implementation of Purdom's algorithm is posted on Prof. Franek's web site (`http://www.cas.mcmaster.ca/~franek`)

Chapter 02

Grammar and Languages

A grammar is a set of construction rules defining which sequences of tokens (terminals) are valid: any sequence built in accordance with the rules is deemed valid, otherwise it is not. A sequence of lexemes is valid if the sequence of corresponding tokens is valid (tokens can be viewed as names of classification groups, while lexemes can be viewed as members of these classification groups – for instance a token INTEGER can have many lexemes, e.g. "2" or "27" or "1234567689"). A programming language defined by a grammar is simply the set of all possible valid sequences of lexemes. Context-free grammars are used to define the syntax of programming languages (which is dealt with by a parser component of a compiler); semantics of the language is beyond the scope of the grammar (and is dealt with by a specialized component of a compiler). A language can be defined by more than one grammar [11].

A formal grammar is defined as four-tuple (N, T, S, P) where N and T are disjoint sets of symbols known as *terminals* (or *tokens*) and *non-terminals* respectively, S is a distinguished element of N known as the *start symbol*. The set of *production rules* $\alpha \rightarrow \beta$ (or *productions*) $P \subseteq (N \cup T)^* \times (N \cup T)^*$ (i.e. $\alpha \subseteq (N \cup T)^*$ and $\beta \subseteq (N \cup T)^*$). ε designates and empty sequence of symbols and a rule $\alpha \square \varepsilon$ is called and ***epsilon rule*** or a ***null rule***.

The language defined by such a grammar consists of all valid sequences of tokens called *sentences*. A sentence is valid if it can be derived from S by a sequence of applications of productions interpreted as left-right rewrite rules (the symbols on the LHS are replaced by the symbols on the RHS). In parsing, we

are interested in "identifying chain of derivation steps that produce valid sequences of terminals strings known as the sentences of the language" [28].

2.1 Different Types of Grammar

It is necessary to define a language in terms of a (finite) grammar -- a relatively small set of production rules, as it is impossible to list or define the potentially infinite set of all valid sentences.

Formal grammars were divided into a number of different classes by Chomsky in 1956. This is known as Chomsky hierarchy [7].

- **Type-0 Grammar (Unrestricted grammar):** is a formal grammar with no (additional) restrictions. Unrestricted grammars define languages that can be accepted by a Turing machine. Such languages are also known as "recursively enumerable languages".
- **Type-1 Grammar (Context-sensitive grammar):** is a formal grammar with production rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where $A \in N$, and $\alpha, \beta, \gamma \in (N \cup T)^*$ and $\gamma \neq \varepsilon$. A rule $A \rightarrow \varepsilon$ is allowed as long as A does not occur on the RHS of any other production. A language generated by a type-1 grammar is a context sensitive language, such languages are recognized by linear bounded automata.
- **Type-2 Grammar (Context-free grammar):** is a formal grammar with production rules are of the form $A \rightarrow \gamma$, where $A \in N$ and $\gamma \in (N \cup T)^*$. The languages generated by context-free grammars are accepted by non-deterministic pushdown automata. Type-2 grammars are the theoretical basis for the syntax of most programming languages.

- **Type-3 Grammar (Regular grammar):** is a formal grammar with productions of the form **A** → **a** or **A** → **aB**, where **A**, **B** ∈ ***N*** and **a** ∈ ***T***. The languages generated by regular grammars are referred to as *regular languages* and are recognized by finite state automata. Regular languages are commonly used to define search patterns in text and lexical structures (tokens) of programming languages. Regular languages can also be defined by regular expressions.

2.2 Ambiguity in Grammar

A grammar that has more than one parse tree (or, equivalently, has two or more derivations) for a sentence is said to be ambiguous. A language generated by an ambiguous grammar is an ambiguous language.

Since grammar plays such an important role in compiler construction, ambiguity of the grammar is undesirable as it causes difficulty in understanding the semantics of the language and causes troubles in parsing (as the parsing must be unambiguous). Thus, ambiguity should be removed from the grammar at an early stage. Unfortunately, the general question of whether a grammar is unambiguous is undecidable, i.e. there is no algorithm that can determine the ambiguity of a given grammar.

Following is an example of an ambiguous grammar (this is a well-know problem of *dangling else*, C stands for *condition*, S stands for *statement*).

S → if C then S
S → if C then S else S

S → other

Consider a sentence **if C then if C then S else S**. It has two possible derivations, **if C then (if C then S else S)** and **if C then (if C then S) else S** .

The degree of ambiguity of a sentence is the number of its distinct parse trees. A grammar has a bounded ambiguity if there is a bound ***b*** on the degree of ambiguity of any sentence of the grammar [10]. Two grammars are said to be *equivalent* if they define the same language. It is often possible to find an equivalent unambiguous grammar, however there are so-called unambiguous *inherently ambiguous languages* for which no unambiguous grammars can be found (for example, $\{ a^n b^m c^m d^n \mid n, m > 0 \} \cup \{ a^n b^n c^m d^m \mid n, m > 0 \}$ is a well-known context-free inherently ambiguous language).

2.3 Syntactic Metalanguage

The notation for defining / describing the syntax of a language in terms of the production rules is called *Syntactic Metalanguage*. Every rule contains a non-terminal and one of its possible terminal string derivations. For a clear formal description and definition, a standard syntactic metalanguage is required. Major functions of a syntactic metalanguage are:

- It names the various syntactic components of the language (i.e.; terminals and non-terminals).
- It describes the valid sequences of symbols (i.e. valid sentences).
- It gives the syntactic structure of any sentence of the language.

In the absence of a standard metalanguage, a programming language definition

must start first by defining the metalanguage, which requires a lot of effort and may causes many problems. There have been a number of syntactic metalanguages used and standardized over the years:

- COBOL (ISO 1989:1985)
- BNF (Used for Algol 60)
- Obsolete FORTRAN 77 (ISO 1539-1980)
- POSIX (ISO/IEC 9945-2:1993)

A syntactic metalanguage should satisfy a number of objectives such as:

1. to be concise: the languages can be defined briefly and can be easier to understand.
2. to be precise and formal: the rules it defines are unambiguous and can be parsed or processed by a computer program.
3. to be natural: the format and notations used are simple to understand for people other than the language designers.
4. to be general: the notation can be used to define different languages.
5. to be simple and self-describing.
6. to be linear: the syntax structure can be expressed as a single stream of characters [37].

2.4 BNF Notation

The *Backus-Naur Form* devised by John Backus and shortly after improved by Peter Naur in 1963 to define the grammar for Algol 60 programming language is the most commonly used syntax metalanguage for context-free grammars. BNF is a formal mathematical way to define the grammar; it can help remove

ambiguity and also can aid in building a parser for the language. There is also an extended version (EBNF) that introduces a better notation for repetitive structures (such as lists) and options, both require inn BNF additional rules.

The meta-symbols of BNF are:

::= meaning "is defined as"

| meaning "or"

< > angle brackets used to surround syntax rule names (non-terminals) as the terminal symbols which are written exactly as they are to be represented.

A BNF rule defining a non-terminal has the form:

non-terminal ::= a sequence consisting of terminals or non-terminals separated by the meta-symbol | .

In some versions of BNF grammar, literal terminals may be enclosed by single quotes, rather than using <> surrounding the non-terminals, or ::= is replaced by the symbol → , this is the version we are using throughout this thesis. White space (blanks, tabs, newlines) is treated differently in different versions of BNF -- some use a special character for it, while others do not.

2.4.1 BNF Example

S → '-' FN | FN

FN → DL | DL '.' DL

DL → D | D DL

D → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Valid sentences generated by this BNF grammar would consist of (unsigned) whole numbers. Below is a sample derivation of 3:

$$S \Rightarrow FN \Rightarrow DL \Rightarrow D \Rightarrow 3$$

2.5 Context- Free Languages

Context-free languages are the most important class of formal languages for both linguistics and computer science. The standard formalization of such languages is based on a rewriting system known as ***context-free phrase structure grammar*** first introduced by Noam Chomsky in 1950's [7] to reconstruct the practice of much earlier traditional and structuralist syntactic description [8]. However, Chomsky introduced the term "type-2 grammar" for context-free languages, the description was discussed above.

The grammars that can be expressed using BNF are exactly the context-free grammars. They are called context-free because the substitution of the LHS symbol of a production by the RHS sequence of grammar symbols of the production is always permitted, regardless of the context in which the symbol is embedded within the sentence [32].

As stated above, the context–free languages are exactly the ones accepted by non-deterministic push-down automata. A non-deterministic pushdown stack automaton is a non-deterministic automaton with a last in, first out (also referred to as stack) memory access.

Below is an example of a context-free grammar (Table 2.1) and the resulting context-free language.

S → A B
S → A S B
A → 'a'
B → 'b'

Table 2.1 Context-free Grammar

The language defined by the grammar of Table 2.1, $L(G) = \{ a^n b^n \mid n \geq 1 \}$. (*Note: this is a well-known example of a context-free grammar that is not regular*).

Chapter 03

Parsing

Syntax analysis (or more commonly *parsing*) is the activity of checking whether a given sentence (in the form of token sequence as generated by the lexical analyzer) belongs to the language and, frequently, generating a parse tree for the sentence. It determines whether the input data (source program) has some pre-determined structure. The parser is the component of a compiler that performs this activity. A schematic architecture of a typical compiler with the major components indicated is given in Fig. 1.

The parsing requires a grammar to be defined (according to which the parsing is performed). The rules of the grammar specify the patterns of valid sentences for the language. Rules can be recursive if they somehow refer back to themselves, in particular left-recursive rule is a rule where the LHS non-terminal and the first (leftmost) RHS symbol are the same.

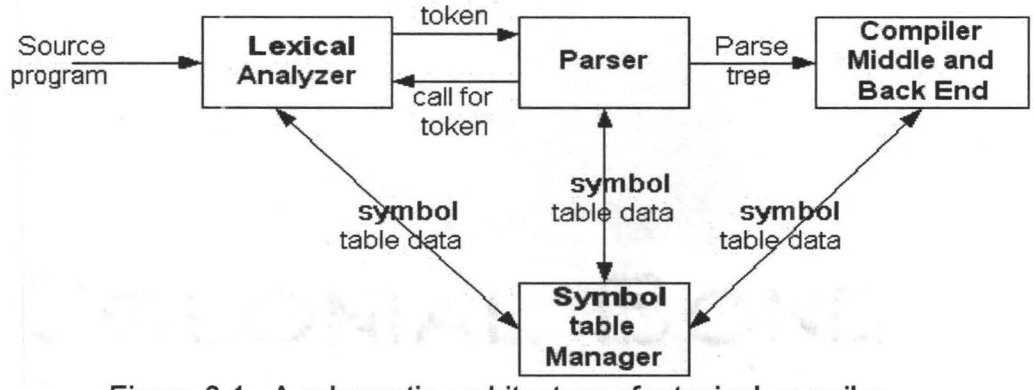

Figure 3.1 A schematic architecture of a typical compiler

In a sense, the parser is the most important component of an interpreter or compiler. It is responsible for performing syntactic analysis on a stream of input tokens. After receiving an input token stream, the parser verifies that it confirms to the syntax of the language or report an error if it does not, moreover it should also recover the errors in order to continue processing. Output of the parser is very often a parse or syntax tree which is used as an input to other components of the interpreter or compiler [30]. There are two main kinds of parsers:

- Top-down parsers
- Bottom-up parsers

3.1 Top-down Parsers

Top-down parsing checks if a sentence belongs to a language by constructing the parse tree from the root (which is the start symbol) and applying productions forward to expand non-terminals into strings of symbols. For every node "n" in the tree, the following two steps are performed by the parser:

- For node ***n***, labelled with a non-terminal **A**, select one of the productions for **A** and construct children nodes of **n** for the symbols on the RHS of the production used.
- Find the next node at which a sub-tree is to be constructed.

The above steps are implemented during a single left to right scan of the input string [1].

Consider the following grammar G:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow \mathbf{num} \mid \mathbf{id}$

where E is the start symbol, E, T, and F are non-terminals and **+** , ***** , **/** , **num** and **id** are terminals. Starting with E and generating a parse tree for a sentence **id + num * id** (see Fig. 3.2) the following sequence of leftmost (the leftmost non-terminal is the one always being rewritten) derivations takes place:

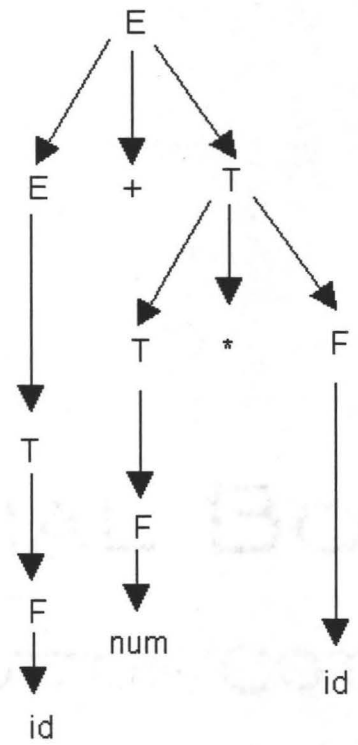

Figure 3.2 Top-down Parse Tree

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow \mathbf{id} + T \Rightarrow \mathbf{id} + F * T \Rightarrow \mathbf{id} + \mathbf{num} * T \Rightarrow \mathbf{id} + \mathbf{num} * F \Rightarrow \mathbf{id} + \mathbf{num} * \mathbf{id}$

While generating the parse tree for the given string, the current token being scanned is referred as the lookahead symbol. Initially the lookahead is the first symbol, i.e. the leftmost symbol of the input. When we get a terminal at a node ***n*** which matches with the lookahead symbol, we progress both in the parse tree and the input. The next token in the string will become our new lookahead and we move to the next child of node ***n*** and so on.

At any stage in top-down parsing, selecting a production rule is a trial-and-error method, if the production gives us the string we have, we select it, and otherwise we backtrack and select another rule. In special classes of grammars, so-called LL(k) grammars, the backtracking can be avoided as by investigating the next k tokens a production can be selected unambiguously, such parsing is referred to as predictive parsing. Very often though, the term predictive parsing is really applied only to parsing of LL(1) grammars.

In the term LL(k), the first L stands for "scanning left to right" (meaning that the input is scanned from left to right to produce tokens), the second L stands for parsing corresponding to the leftmost derivations, while k stands for the number of token lookahead.

Top-down parsers are easier to code manually and to debug. They have smaller code and can include the lexical analyzer and hence tokenize quickly. On the other hand, they are slow in backtracking and are unable to handle left recursive rules of the form $\mathbf{A} \rightarrow \mathbf{A}\,\alpha$.

3.2 Predictive Parsing

It is better to design a grammar for which the parser does not have to use

backtracking or large lookahead. Top-down parsing without backtracking and lookahead is known as predictive parsing. It can only to a special class of grammars. Such grammars must have the following features such as:

- given an input token **a** and a non-terminal **A** to be replaced, it can be determined unequivocally which production **A** will lead to a string beginning with **a**.
- no two productions lead to strings with the same starting terminal symbol for the same input token to avoid lookahead and backtracking.
- No production is has left-recursion, as it would lead to infinite loops in parsing.

Predictive parsing is best to use for languages with keywords such as **if**, **while** or **begin** which immediately identifies the construct. If a grammar is not suitable for predictive parsing, we may be able to transform it to an equivalent form that might be more suitable. Since ambiguity in grammars will lead to duplicate entries in predictive parsing tables; we have to remove ambiguity from the grammar to do predictive parsing.

3.2.1 LL(1) Parsing

LL(k) parsing for larger k's is generally not use in practice because the slowdown of such parsers grows exponentially with k. Therefore we are confining ourselves to only LL(1) parsing and LL(1) grammars. The MACS grammars used in our thesis were all LL(1).

There are two ways to implement LL(1) parsers, as a recursive descent program (and that is what JavaCC provides), or a non-recursive implementation

using a parsing table.

For recursive descent parsers, the principle of turning the productions to code is fairly simple and straightforward: each non-terminal corresponds to a procedure, each rule with non-terminal **A** as its LHS is interpreted as a part of definition of the procedure **A**. For each symbol of **α** of the rule $\mathbf{A} \rightarrow \boldsymbol{\alpha}$, a terminal corresponds to looking at and/or consuming the next token, while each non-terminal corresponds to calling the procedure of that name. Thus a rule $\mathbf{A} \rightarrow \mathbf{B\ b}$ ... becomes a "definition" of the procedure **A** that first calls procedure **B** and then calls for next token and checks whether it is **b** ...

The non-recursive LL(1) parsers consist of a parsing table, a stack, and input buffer with the sentence to be parsed. The parsing table is pre-computed, its columns are labelled by terminals (including a special symbol indicating the end of input – often the null character if the input is in the form of C strings), its rows are labelled by non-terminals. Each entry in the table for a terminal ***t*** and non-terminal **A**, is either ***error*** or a single production $\mathbf{A} \rightarrow \boldsymbol{\alpha}$. The meaning and use of the table is: "if non-terminal **A** is on the top of the stack and if current token is ***t***, if the table entry at column ***t*** and row **A** is ***error***, then the string being parsed is syntactically incorrect, otherwise use the production $\mathbf{A} \rightarrow \boldsymbol{\alpha}$ in that entry to rewrite **A** by its RHS (i.e. pop A from the stack, and push on the stack one by one the symbols of **α** in reverse order).

Initially, the stack contains at the bottom the end-of-input terminal and the start non-terminal. The whole parsing algorithm can be summarized as:

- Let **A** be at top of the stack and **a** be the current input token, if $\mathbf{A} = \mathbf{a} = \boldsymbol{\varepsilon}$, the parser stops.
- If **A** is a terminal, pop the stack and move to the next input symbol.

- If **A** is a non-terminal, use the current token to lookup in the parsing table what to do (either ***error***, or a production to be used).

The parsing table is built using two functions related to the underlying grammar G. They are:

FIRST(**A**): Let **A** be a string of grammar symbols, FIRST(**A**) is the set of terminals that begin the strings derived from **A**. If there is a rule $\mathbf{A} \Rightarrow^* \varepsilon$, then ε will be in FIRST (**A**) as well, i.e. more formally FIRST(**A**) = $\{ \mathbf{a} \mid \exists \alpha \; \alpha \Rightarrow^* \mathbf{a}\alpha \}$

FOLLOW(**A**): For non-terminal **A**, it is the set of terminals that can appear immediately to the right of **A** in some sentential form, i.e. more formally FOLLOW(**A**) = $\{ \mathbf{a} \mid \exists \alpha, \beta \; S \Rightarrow^* \alpha A a \beta \}$ [32].

3.2.2 LL(1) Grammars

Now we can formalize the definition of LL(1) grammars. Set

$$\textbf{Lookahead}(A \rightarrow B_1 B_2 \ldots B_n) = \bigcup \{ \text{FIRST}(B_i) \mid B_i \Rightarrow^* \varepsilon \} \cup X$$

where X = FOLLOW(**A**) if $B_1 B_2 \ldots B_n \Rightarrow^* \varepsilon$, otherwise X is empty.

A **grammar *G* is LL(1)** if for any two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ such that $\alpha \neq \beta$, Lookahead($A \rightarrow \alpha$) $\cap$ Lookahead($A \rightarrow \beta$) = $\varnothing$.

A grammar can be LL (1) if and only if the following conditions hold: whenever two distinct productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ exist, the following properties hold:

- For any terminal symbol **a**, both α and β should not derive strings that begin with **a**.

- At most one of **α** and **β** can derive an empty string.
- If **β** $\Rightarrow^*$ **ε**, then **α** does not derive any string that begins with a terminal from FOLLOW (**A**). [32]

Let us consider a grammar ***G*** with the set of terminals ***T*** = { **a**, +, *, (,) } and the set of non-terminals ***N*** = { T, E, F } with E being the starting non-terminal, and the set of production rules given as: ***P*** = { E → T | E + T, T → F | T * F, F → **a** | (E) } . If we generate the parsing table for this grammar, it will contain duplicate entries, or equivalently the Lookahead sets will not be disjoint :

Lookahead (E → T) = { **a**, (}
Lookahead (E → E + T) = { **a**, (} (*conflict with previous*)
Lookahead (T → F) = { **a**, (}
Lookahead (T →T * F) = { **a**, (} (*conflict with previous*)
Lookahead (F → **a**) = { **a** }
Lookahead (F → (E)) = { (}

Therefore the grammar ***G*** is not an LL(1) grammar. However, we can convert it to an equivalent LL(1) grammar ***G'*** : ***T'*** = { **a**, +, *, (,) }, ***N'*** = { T ,E ,F ,T', E' }, ***P'*** = { E → T E', E' → + T E' | **ε** , T → F T' , T' → * F T' | **ε** , F → **a** | (E) }. For this grammar ($ denotes the special end-of-input symbol):

Lookahead(E → T E') = { **a**, (}
Lookahead(E' → + T E') = { + }
Lookahead(E' → **ε**) = {) , $ }
Lookahead(T → F T') = { **a**, (}
Lookahead(T' → * F T') = { * }

Lookahead(T' → ε) = { +,) , $ }

Lookahead(F → a) = { a }

Lookahead(F → (E)) = { (}

3.3 Bottom-up Parser

The bottom-up parsers are also known as LR parsers. Parsing starts from some pre-defined state and moves to another state (or stays in the same state) depending upon the next available token. If the parser ended in the some pre-defined state, parsing is successful otherwise it signals an error. Bottom-up parsers are usually implemented as a series of states, encoded in lookup tables.

Bottom-up parsers are fast and can handle left recursion. They can be used for parsing of a larger class of grammars and translation schemes (see Fig. 3.3 below), so software tools for automatic generation of parsers from grammars tend to be more of the bottom-up variety.

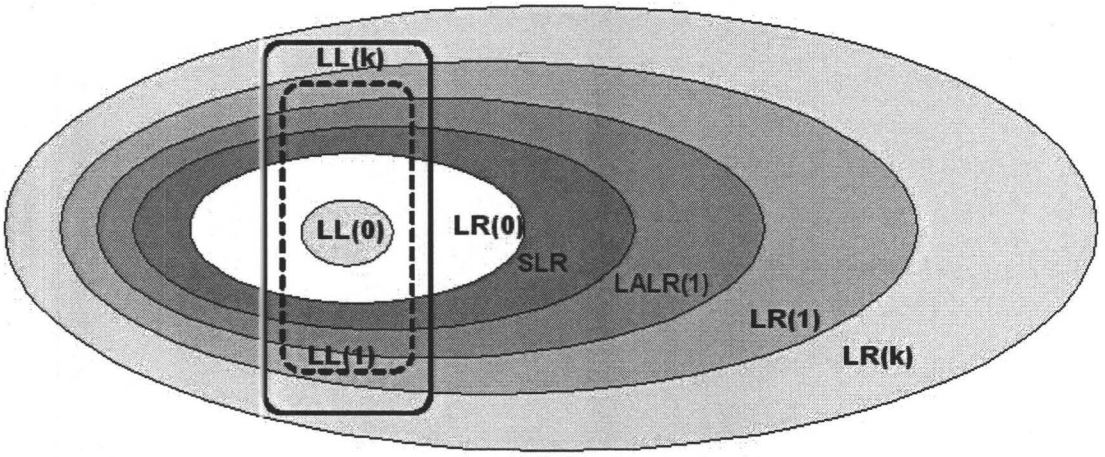

Figure 3.3 Hierarchy of unambiguous grammars

Bottom-up parsers also have a lot of disadvantages such as:

- They have fixed tokenization (i.e. it is virtually impossible to have the scanner and parser built as a single component, while top-down parser often can).
- They are extremely hard to debug and the code size is very large.
- Tail recursion is handled very poorly and inefficiently.
- Cannot predict the execution of semantic actions.

3.3.1 LR Parsing

LR parsers use the shift / reduce technique. Their major disadvantages encompass (a) the construction of the parsing table and (b) the size of the parsing table. All typical LR parsers have the same architecture, only the parsing table is language specific. A typical LR parser consists of an input buffer, a stack, stack table, a parsing table, and an output buffer. The stack table contains actions for every terminal in each state and the **goto** statement for the non-terminals. The stack contains pairs consisting of **value** and **state**; initially the stack table has state 0. The action table for each state has four possible values:

1. Shift and move to state ***n***
2. Reduce using rule number ***n***
3. Error
4. Accept

The parser reads the current state (at the top of the stack) and the current terminal (the next available terminal). It looks for the associated action for the terminal and the state.

- If the action is shift, the parser will push the current terminal and the new state onto the stack.
- A reduce action will pop a suitable number of symbols off the stack, make the state the one now on top of the stack and push the non-terminal of rule ***n*** on the stack, followed by the state specified for the non-terminal into the ***goto*** part of the stack table.
- Error will cause the parse to move in error handling state.
- Accept, the parser will accept the grammar.

Figure 3.4 below depicts a typical architecture of an LR parser.

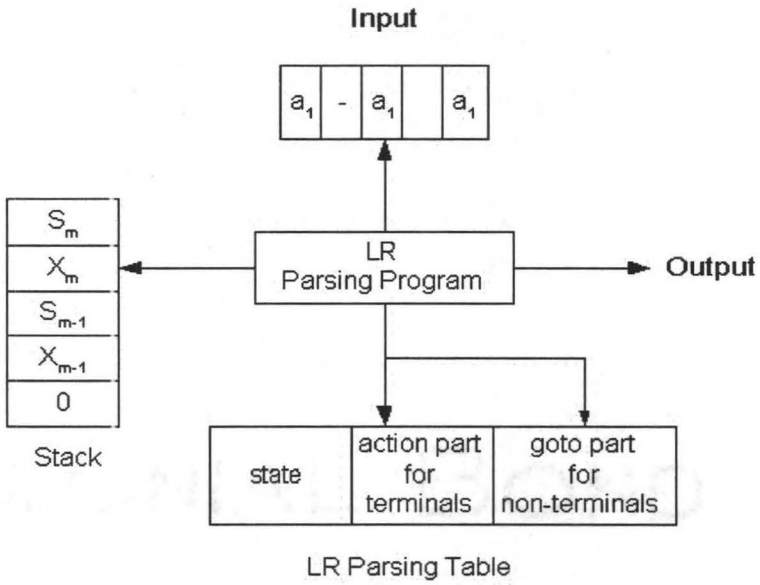

Figure 3.4 LR Parsing architecture

3.3.2 LR Grammars

LR(k) grammars that can parsed by an LR parser. The first L in the designation means "scanning the input from left to right", while the second R in the designation signifies that the parsing follows the rightmost derivation in reverse, and k again signifies the number of lookahead tokens. The LR(1) class of grammar is bigger and includes all LL(1) grammars (see Figure 3.3) . Most of the programming languages have LR (1) grammar [11]. LR grammar define more languages in comparison to LL grammars, because they have more stringent requirements for selecting the production rules: in LL(k) parsing we select the rule by looking the first k tokens whereas in LR(k) parsing, the selection is postponed till we have seen all of what is derived from that right side with k input tokens [32].

Chapter 04

Compiler Validation

The algorithmic aspects of compilation (termination and complexity) have been well studied, but not much attention is paid to its semantic correctness, the fact that the compiler should preserve the meaning of programs. In other terms, the correctness of compilers is generally established only through validation testing. This is adequate for compiling low-assurance software: what is tested is the executable code produced by the compiler, therefore compiler bugs are detected alongside the application bugs. This is not adequate for high-assurance and/or critical software which must be validated using formal methods; for such software, the source code of the application is verified. Therefore, any bugs in the compiler used to transform the source code into the executable module can invalidate the guarantee obtained by formal verification of the source. To establish strong guarantees that the compiler can be trusted not to change the behavior of the program (i.e. its semantics), it is necessary to apply formal methods to the compiler itself. Several approaches in this direction have been investigated, including translation validation, credible compilation, proof-carrying code, and type-preserving compilation.

4.1 Translation Validation

Compiler verification is a complex task as it provides a proof in advance that the compiler always produces a correct output that implements the source code. However, it also discourages even minor modifications of the compiler, since with every change regardless its size, the proof obligations must be re-established. This may in fact impede the compiler design [27].

Translation validation proves the correctness of each individual compilation rather than the correctness of the compiler itself. Each individual translation is followed by a validation phase which confirms that the code produced implements the source language correctly. Research shows that proving the correctness of compilations is a far more tractable problem than proving the correctness of the compiler itself. The validation tool produces a proof script after every run of the compiler. The proof generated by the validation tool can be checked independently, for even greater assurance, by existing proof checkers [36].

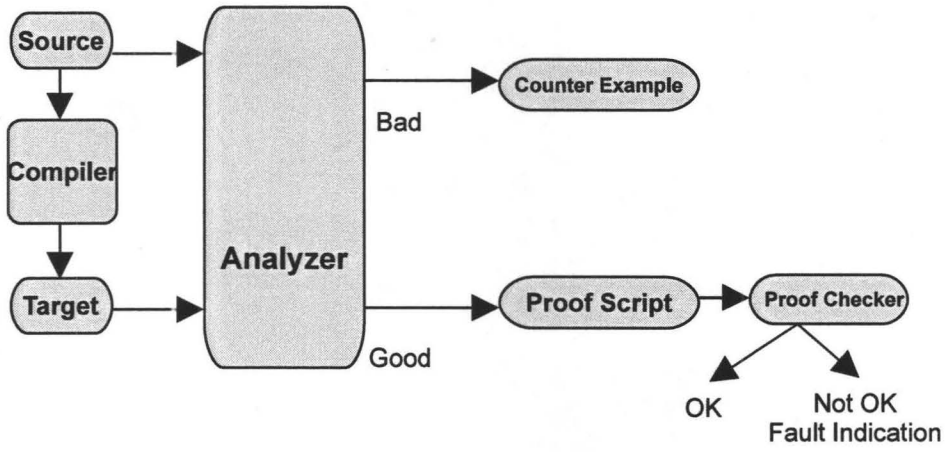

Figure 4.1: The concept of Translation Validation

The whole process of translation validation is shown in Figure 4.1 (see [27]). The source and target programs are given as input to the analyzer which compares them and either generates a proof script if it finds proper correspondence between the two and will gives a counter example if both the two input doesn't

agree. The counter example gives the scenario in which both the two inputs are different and signals an error in the compiler. The proof script generated will then be tested by a proof checker to provide complete guarantee of the compiler.

The framework needed to fully automate the translation validation process must include [27] :

1. A common semantic framework for representing both the source code and the generated target code.
2. The notion of "correct implementation" must be formalized as a refinement relation based on the common semantic framework.
3. A syntactic-simulation-based proof method which can be automated to verify that the produced output implements the source code properly, by comparing the models of target and source codes.
4. Automation of the proof generation method which should successfully generate a proof script.
5. An additional proof checker which examines the generated proof script and gives the final confirmation of the translation.

The validation task of transformation is influenced by translators and thus is becoming more and more difficult with the growing complexity and availability of optimization methods used by the translators. A tool developed for translation validation called CVT (code validation tool), managed to automatically verify translations involving about 10,000 lines of source code in about 10 minutes [33]. However, the success critically depends on restrictions such as source and target programs with single external loop, and allowing a very limited set of optimizations.

4.2 Translation Validation of Optimizing Compilers

The compilation cycle begins when the compiler receives a source program, it then translates it into an intermediate code, the compiler then applies a series of optimizations on this code, starting with architecture independent optimizations (such as common expression elimination, loop-invariant hoisting, etc.), and then architecture dependent ones (such as register allocation and instruction scheduling). These optimizations usually take up to 15 passes in some compilers. Translation validation provides either a proof script confirmation or an unsuccessful validation with a counter example after each optimization pass. Simulation is used to confirm that the general approach of showing the correct correspondence between the target and source code is based on refinements. A refinement mapping is established to show how the relevant variables of source code correspond to appropriate target variables or expressions. Proof obligations are developed for each such refinement. Sometimes it is necessary to introduce auxiliary variables at selected points in the program. The proof obligations are then shown to be valid under the assumption of the auxiliary invariants.

Using the formalism of Transition Systems (TS's) (see [33]), this strategy in general terms is the first to give common semantics to the source and target codes. Every computation of T corresponds to some computation of S with matching values of the corresponding variables is the statement of the notion of refinement, of a target code T being a correct implementation of a source code S. In Figure. 4.2, the process of refinement is presented as completion of a mapping diagram [33] [34] [35].

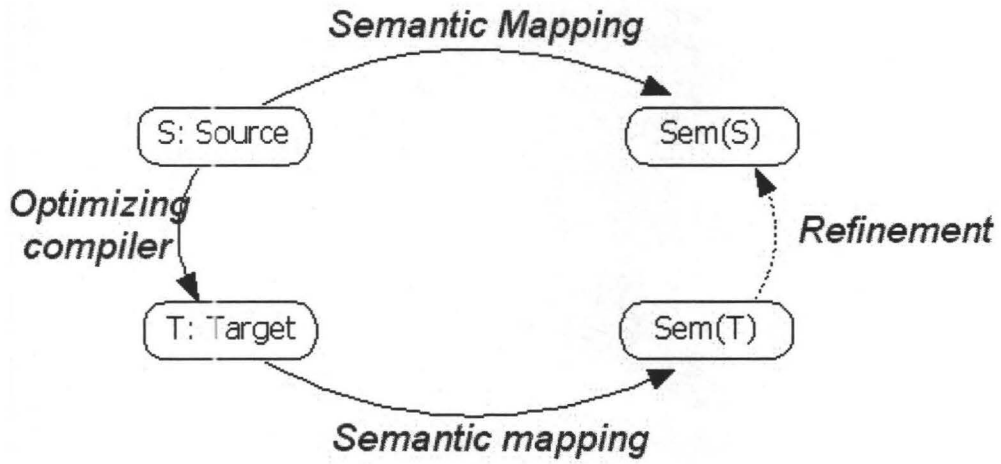

Figure 4.2: Refinement as completion of Mapping Diagram

In some debugging modes, supported by most compilers where only minor optimization or no optimization is performed, the proof that the target code refines the source program is reduced to the proof of the validity of a set of automatically generated verification conditions; proof obligations [34][35], which are implications in first order logic. In such cases we are required to establish the validity of the set of verification conditions only. The proof obligations are in a restricted form of first order logic called *educational formulae*, using uninterrupted functions to represent all arithmetical operations under the realistic assumption that only restricted optimization is applied to arithmetic expressions. Research has been conducted to show the feasibility of building a tool for checking the validity of such formulae. Such tool is based upon finding small domain instantiations of the educational formulae and then using BDD[1]-based representation to check for validity [35]. With the optimization turned on, the validating tool will need additional information specifying which optimizing transformations have been applied in each translation. This additional information

[1]BDD = Binary Decision Diagram

can be provided either by the compiler or can be inferred by a set of heuristics and analysis techniques. Essential information can be provided in the form of program annotation which can be used by the validation tool to form invariant assertions at selected control points. Loop tiling, loop distribution and fusion, and loop interchange are structure-modifying optimization techniques and are more challenging category of optimizations. Since there are often no control points where the states of the source and target programs can be compared, it is often impossible to apply the refinement-based rules for this class. Reordering transformations are the permutation rules which can be defined for a large class of these optimizations that allow for their elective translation validation. The structure preserving methodology can deal with loop unrolling, however loop unrolling naturally falls into the category of reordering transformation and can be dealt with by the permutation rule.

4.3 Validation Test Approach

Compiler validation is done to confirm that the compiler implements the particular programming language correctly. Standard tests are available for testing programming languages such as FORTRAN, Algol, COBOL and Pascal, and in particular ADA. While developing a standard set of validation tests, there are a number of issues to be taken in account [12]:

- How many tests should be enough? Should a few, fairly large tests be sufficient, or a lot of small tests?
- How to minimize the effort needed to test a compiler using different test structures?
- What is the best way of designing high quality tests and what are good measures of validation test quality?
- A test might serve different purposes, which of those are to be

emphasized depending upon how the tests are structured and used?

- How to consider the variety of implementation options permitted by the standards while designing tests, such as the number of levels of numeric precision and the ranges of values associated with each level?

Addressing some of the above given issues:

Number and size of tests: There are two different approaches while considering the number and size of tests, each has its own advantages and disadvantages.

- **Few large tests:** It is easier to prepare and submit few large tests to the compiler. Once the validation test is submitted the team has to wait for its completion. They do not have to go through the burden of submitting a number of small tests to the compiler, get the results, record the results, and then repeat the same cycle for another small test. Often there are new tests to track and modify, and with few large sets this task is simplified. The results can be analyzed more efficiently as all of them are available at the same time. In addition, the processing overhead would be smaller. The biggest disadvantage is error tracking. Besides that, any error (deliberate or otherwise) in such a test may cause a failure to compile the test program. This eliminates all of the negative cases from inclusion into the large test. Moreover, this approach requires a fairly extensive amount of manual intervention during at least the first attempt to validate the compiler.
- **Many small tests:** The main advantage of using many small, mutually independent tests where each test is responsible of validating of a single feature of the language, lies in the fact that we can use a layered approach to the testing. Whenever new features are added to the programming language, new testes are developed and are added to the existing suite as a new layer. In this case the first layer will do the

necessary setup and then call smaller tests themselves, which are considered as a second layer. Failure of any one of the tests does not have any effect on the usability of the others. This eliminates the validation of the test suite altogether and all the effort can be directed to the validation of the compiler. The major disadvantage of having a large number of smaller tests is the failure to test the compiler's code-handling capacity (i.e. whether the compiler is capable of executing large programs or programs with large number of symbol references).

Classes of tests: Tests are classified according to the general nature of their criteria for passage or failure. This classification has been used for validation of ADA compilers and is generally recognized as *ADA Conformity Assessment Test Suite* (ACATS) [12]. According to ACATS, there are six classes of tests:

1. **Class A** tests are passed if no errors are detected at compile time. Although these tests are constructed to be executable, no checks can be performed at run time to see if the test objective has been met; this is what distinguishes Class A tests from Class C tests. For example, a Class A test might guard against superset implementations by checking for keywords of other languages (those not already reserved in the source language), to ensure that they are not treated as reserved words by the compiler being tested. Although execution of such a test sheds no additional light on whether the test has been passed, it is usually convenient since all other tests (except Class B tests) are executable.

2. **Class B** tests are negative tests which contain illegal statements. They are considered passed if all the errors they contain are detected at the compile time. They do contain some legal statements which should compile without problems with the compiler.

3. **Class L** tests consist of illegal programs whose errors need not be detected until link time. They are passed if errors are detected prior to initiating execution of the main program.

4. **Class C** tests consist of executable self-checking programs. They are passed if they complete execution and do not report failure.

5. **Class D** tests are capacity tests. Since there are no firm criteria for the number of identifiers permitted in a compilation, the number of units in a library, etc., there are no clear pass/fail criteria.

6. **Class E** tests are constructed when ambiguities are discovered in the standard; they determine how an implementation has interpreted the ambiguity. The results of these tests do not determine the validity of a compiler, but provide information that helps the users of the compiler and the government keep track of how implementers are "voting" on the interpretation.

It has been noted that negative tests (i.e. tests which deliberately contains errors in themselves) have proved to be more valuable in detecting compiler deviations; it has been reported [12] that more implementation errors were detected with Class B tests than Class C tests. Of course, illegal programs are needed to detect supersets. The above classification of tests indicate the breadth of test coverage, thereby helping the automation of test results analysis.

In practice it is impossible to write test cases which are same for all compilers and detect all possible errors. Writing test cases for compiler should

undertake the knowledge of the code implementing the compiler. With these resources limitation, we should concentrate on writing such test cases which when passed give the user the confidence that the remaining errors will only rarely occur in practice.

The validation process

The main objective in compiler validation is to design test, tools and procedures to minimize the manual effort needed to validate a compiler. The compiler validation procedure consists of the following steps:

- Gather the data needed to customize the test for the compiler under validation.
- Generate the implementation-dependent versions of test. The file names and tests should be according to the compiler's environment and so as the commands needed to submit the tests to the compiler.
- Compile and execute the tests and collect results for future analysis.
- Analyze and summarize the test results.
- Document all the results and any special tool used for testing the compiler. This information would be useful when the compiler will be retested.

Three different environments are present for validating compilers.

- **Validation environment**
  Tests are prepared for execution and validation results are analyzed.
- **Compiler environment**
  Tests are compiled and the executable code is then transported to the target computer for execution. In some implementations separately compiled codes are also linked with the tests.

- **Execution environment**
  Test programs are executed on the target computer

The validation environment supports all testing activities. There are specialized tools available to help reduce the efforts required to perform a validation.

4.4 Certifier Approach

Certifying compiler is a method to ensemble the compiler and the certifier together, so that the optimizing compiler will translate a strongly typed programming language into assembly language program and the certifier will either produce a formal proof of type safety or a counter example pointing to a potential violation by the assembly language target program [25], [9].

The method has a number of advantages:

1. Easier to implement as compare to formal verification of the compiler, as most of the compiler changes do not require to change the certifier and also verifying the correctness of the result is easier than to verify the correctness of the computation.
2. The method can be applied to any optimizing compiler as the optimization is independent of the certifier design.
3. The method can be applied to certify other properties besides type safety of target language and can be applied to the compilation of any type safe language.
4. It improves the effectiveness of compiler testing as it confirms the

correctness of each compilation and for each test case ,it statically signals compilation errors which otherwise might take several executions to detect.

5. The most practical method to automatically generate the safety proofs for a proof carrying code system for type safety.

From an abstract point of view, the certifying compiler is like a pipe comprising a compiler and a certifier. The compiler produces the assembly code along with the code annotations and type specifications. Other tasks done by the compiler includes global register allocation with spilling and coalescing which causes a register within a single code block to be used to store different values types, and global optimizations. Therefore to verify that target programs are memory safe and type safe is very difficult. Code annotation helps the certifier to understand the code and not to pay much attention to the optimization to verify the memory and type safety.

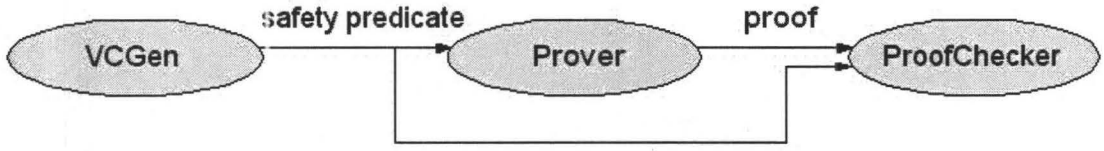

Fig. 4.3: The Overview of a Certifying Compiler

The certifier subsystem is itself a combination of three subsystems: the verification condition generator (VCGen), the prover, and the proof checker, as in Figure 4.3 (see [25]).

The VCGen can be executed on a function-at-a-time basis and can be implemented as an efficient single pass through the program due to the code annotations and typing specifications. It generates safety predicate for each function in the code, each of these predicates have proofs if the functions are type and memory safe. They use code annotations and type specifications while scanning the assembly language program. The prover uses the first order logic predicate and produces a formal proof.

In the last stage, a simple proof checker takes the safety predicate and the resulting proof as input, to verify the validity of the proof against its safety predicate and to judge that the compiler output is memory-safe and type-safe. Hence for a system that uses Proof-Carrying Code to enable the safe execution of entrusted mobile code, certifying compiler can serve as an automatic front-end.

Chapter 05

Compiler Test Case Generation Methods

5.1 Compiler Testing

A compiler is a computer program that accepts a source program and produces either compiler error messages or an object code corresponding to the valid source program. Due to the complexity of the compiler as a program, checking the conformity of a compiler to its specifications is a complex task. Since compilers are frequently used, their verification is critical for the correct creation and execution of other programs. Prior to its release, it is tested to show that it correctly implements the particular programming language.

The aim of compiler testing is to verify that the compiler implementation conforms to its specifications, which is to generate an object code that faithfully corresponds to the language semantic and syntax as specified in the language documentation. Finding an optimal and complete test suite that can be used in the testing process is often an exhaustive task. Various methods have been proposed for the generation of compiler test cases. A lot of research has been done on testing compilers, most of which has addressed compilers for classical programming languages such as Fortran, Pascal, or Algol.

The most commonly used technique for testing compilers is *functional testing*. A series of independent test cases are designed to test individually all the functional features of the language. Each test case is designed to test a limited functionality of the language, which simplifies the process of testing by focusing on a single objective and minimizing the interactions between the language

features. Unexpected interactions between the different features cause compilation problems. In the following sections, we will classify different types of compiler testing.

5.2 Different Types of Compiler Testing

Testing is the process of finding errors by executing a program. Testing does not ensure the absence of bugs in a program, nevertheless, it signals the presence of such. This helps develop some degree of confidence that the compiler behaves correctly for some input data. There are two different test execution strategies used for testing software.

- In **Static Testing,** the source code is inspected without running it. The code can be read or reviewed for error without executing it by the developer itself. It mainly checks the syntax of the code.
- **Dynamic Testing** is a process of finding errors by executing the software using a set of test data in a controlled test environment. The actual outcome is then compared with the expected outcome.

Compilers are tested mostly using the dynamic strategies. There are two different categories of test cases for the dynamic test execution strategy

- **White Box Test Generation**

White box testing is done based on the complete knowledge of the internal structure of the program. Test cases are developed with prior knowledge of the implementation details. Test data are generated on the basis of program logic, structural control, and using data flow techniques. Test cases are prepared for each transition and state change in the code. The test cases generated are in

close correspondence with the code, so minor changes in the code require changes in the test cases as well. This technique is also known as *Glass box testing*. Frequently used methods that fall under the white box purview are: logic coverage testing, statement coverage, and decision coverage.

- **Black Box Test Generation**

In black box testing it is the desired output that is verified; the test data are generated without the knowledge of the actual implementation details of the code solely from the software specifications. Its main focus is on the program features and its external behavior. In case of compiler testing, this type of testing is done to confirm certain features of the compiler according to the language specifications. It is used to certify the conformance of the compiler to the language standard definition, which is an increasingly important issue in the marketability of the compiler. Frequently used methods that come under black box test purview are: specification-based testing, equivalence partitioning, and boundary value analysis.

In practice, it is almost impossible to perform exhaustive white box or black box testing. It is a better approach to develop a reasonable testing strategy that makes use of both techniques. A strict test can be develop using certain black box oriented test case design methodologies and then supplementing these test cases with white box oriented methods [4] [3] [13].

5.3 Selection of Testing Method

While selecting a method for compiler testing, we have to keep these issues under consideration:

- **Test Case Generation Strategy:** Selection of the systematic method of

test cases.

- **Test Selection Strategy:** Selection of subset of generated test cases. Test cases which reveal maximum errors can be selected from a large set of automatically generated test cases.
- **Test Execution Strategy:** Suitable strategy for executing the selected test cases.
- **Test Specification Language:** The language used for the formal description of the test cases.
- **Test result Analysis:** How the output obtained after the test and the expected output be compared, analyzed and a test verdict is obtained.
- **Test Coverage and Metrics:** The extent to which the software functionalities are covered. Ideally test cases should cover all the syntax and semantic details of the compiler under test and produce all possible compile and runtime errors.
- **Test Case Correctness:** The verification of the validity of the test case design.

5.4 Test Case Generation Methods

The creation of an effective set of tests can be a substantial task involving the analysis of a thousand combinations of cases, to develop manually such test cases for all possible combinations of the language features is very laborious. A possible solution is to find ways how to generate these cases automatically. In the context of compiler testing, according to [24], a test case consists of:

1. A test purpose or test case description.
2. A test input consisting of a source program for which the behavior of the compiler under test is verified.

3. An expected output which may include a reference to an output file or error file.

When test cases are executed, they should give clear and unambiguous results. They should be complete, i.e. should cover all the syntax and semantic details of the language and should signal all possible type of errors. The testing process starts with a grammar which is an input to a program generator. The generator creates test programs and the expected output. The test program is then processed by the compiler and the compiled code is executed. The actual output and the expected one generated from the source code are then compared. The compiler is said to have passed the test case if both of these output matches, and, if applicable, the error messages match the expected error messages as specified in the test case expected output details. The different stages of compiler testing are shown in Figure 4.1 [5].

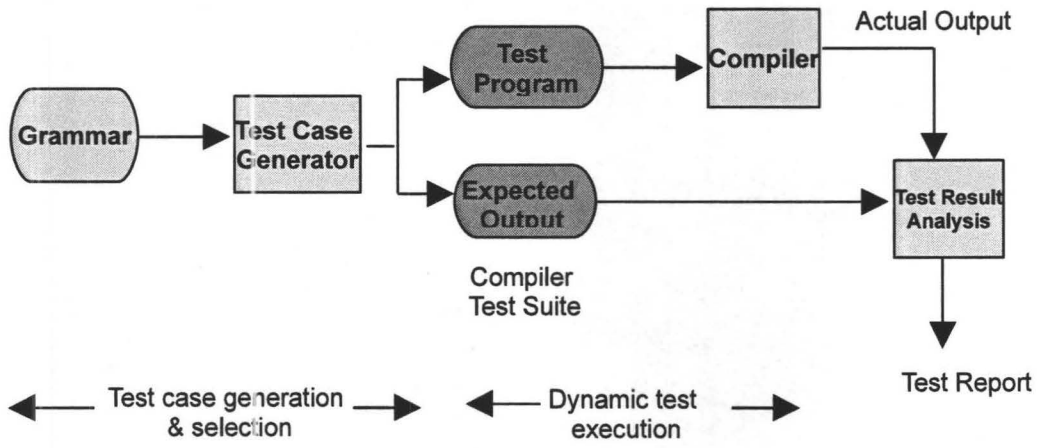

Figure 5.1 Phases of compiler testing

5.5 Assessment Criteria of Test Case Generation Method

Various methods are available for the generation of test data for compilers. There are a number of metrics on the basis of which we can decide which method is best to use. They are:

- **Type of Grammar:** Different generation methods are available for context-free, regular dynamic, attributed, enhanced context-free and context-free parametric grammars.
- **Data Definition Coverage:** The test cases developed should cover all the data definitions defined in a programming language.
- **Syntax Coverage:** The generated test case should cover all the syntax of the language for which the compiler is written. The generator should produce both the syntactically correct and incorrect codes.
- **Semantic Coverage:** It should cover all the semantics of the language and should generated both semantically correct and incorrect programs.
- **Extent Of Automation:** The testing of compliers is the most suitable automation area, as the test case generation methods should be relatively easy to automate as the test data have very good specification (in the form of the grammar).
- **Type of Language:** The methods can be used for a range of different languages.
- **Implementation and Efficiency:** The method should be efficient as well as easy to implement.
- **Test Case Correctness:** The method should generate correct test cases.
- **Concurrency en Exception Handling:** It should be applicable to compilers for concurrent languages and should cover the exception handling feature of the underlined language.

5.6 A brief Survey of Test Cases Generation Methods

- **1970, Hanford:** He defined a *syntax machine* which automatically generates random test cases for any programming language. However, he implemented it with a dynamic grammar (context-free grammar that can modify itself) to generate data for PL/1 compilers. The method produced meaningless yet syntactically valid programs. It concentrates on modular and procedural languages like FORTRON and PL/1. Compiler reliability related to problems such as infinite loops, abnormal termination or diagnosing non-existent syntax errors can be checked [13].
- **1972, Purdom:** He used a syntax-directed method to generate test sentences for a parser. Purdom's algorithm generates small sentences efficiently with the goal to use each production rule at least once [29].
- **1974, Seaman:** The main idea was to compare the results. Programs were compiled on different compilers and the results were then matched. The test program was first executed on the checker with the WRITE statement writing all the variables and then the values were read by running READ operation with the optimizer. Any change in the values would signal an error. Decisions were based on some biasing factors. Changing them would affect the area that was in the focus. Another approach was the use of program generator, which makes decisions on the coded source constructs on the basis of pseudo-random numbers. The generator keeps track of the values of variables declared in the program it is generating, and generates comparison operation between these variables and the constants that represent the values the variables ought to have [5].

- **1976, Wichmann and Jones:** The method proposed by them can detect any significant error in the syntax, but is unable to do semantic testing. They used a large set of small programs, with each of them having some unusual features to test the rarely executed parts of the compiler code. Two different types of tests are used: *exhaustive tests* that check every part of the compiler, and the second type of tests will ensure the size limits, these tests are not exhaustive. The method can check the depth of nested constructs such as procedures, loops, and blocks. The tests cases ware executed on four different Algol 60 compilers [5].

- **1978, Duncan and Hutchison:** An attributed-grammar based method was proposed for generating semantically correct test data. Test cases were generated to test how the program handles certain classes of input data instead of checking the sections of code. Test cases were based on program specification. Context sensitive information needed to generate semantically correct data was provided by the attributes. The method also makes a selection of test cases according to various criteria. Passing information during the test cases generation becomes explicit by including an appropriate attribute in the grammar [5].

- **1979, Bauer and Finger:** They used regular grammar to generate complete test cases for finite state control programs. In this system they used an augmented finite state automaton model (FSA). A test sequence was generated to test the system thoroughly by giving the FSA model. The test cases described a sequence of stimuli to be applied to the system under the test and the corresponding required results. The test case generation scheme was implemented in a component called the *test plan*

*generator (*TPG*)*.The number of test sequences generated depended upon a number of factors such as: number of function states, number of stimulus types, and the cyclic nature of the system specification [5].

- **1980, Celentano et al:** An automatic sentence generator was defined on the basis of the language to be compiled. The language definition was given by a grammar in extended BNF, which was further augmented with actions to ensure contextual harmony e.g. between data definitions and use of identifiers. First, all sentences correct with respect to the given context-free grammar were generated to verify the syntax analyzer (parser) of the compiler. Purdom's algorithm was used to generate the sentences. To further test the deep control structure of the grammar, the sentences went through stepwise refinements, resulting complete compilable programs. The refinements are controlled by rewriting rules enriched by a parameter passing mechanism. The generator was tested with PLZ, MINIPL, and some other languages [6].

- **1982, Bazzichi and Spadaforahi**: The compiler was tested by compilable programs which were generated automatically by a test generator. The main idea of the generator was to produce programs with all the grammatical constructs of the source language. The methods generated both correct and incorrect programs to check the performance and efficiency of different compilers for the same language. The input to the generator is a grammar given in a tabular form. The generated test programs cold check different parts of the compiler, such as the lexical analyzer (scanner), the syntax analyzer (parser), the semantic analyzer, the diagnostic and the error handling routine. Some significant results were obtained for Pascal [2].

- **1985, Mandl:** This method was used successfully in designing some of the tests for Ada compiler validation capacity (ACVC) test suites. It is a method which yields the informational equivalent of exhaustive testing at a fraction of the cost for testing compilers. Random selection of test cases is a better approach than the exhaustive testing. In order to achieve the high level of conformity the non-exhaustive test procedure should be selected carefully. It was made explicit what conditions would render such a procedure satisfactory - perhaps even as satisfactory as the exhaustive testing. The method proposed is one that used the properties of orthogonal Latin squares (a special kind of combinatorial designs, an n $\times$ n Latin square is a 2-dimensional integer array where each row has entries 1, 2, ..., n and each column has entries 1, 2, ..., n). For ***k*** variables each admitting n values, choose a set of ***k*** - 2 orthogonal n X n Latin squares and implement that. Instead of the total number n^k of test cases, only n^2 combinations are needed [21].

- **1989, Homer and Schooler:** A test case generator TCG for large compilers whose modules communicate through complex graph structured intermediate representation. The input to the generator was a context-free grammar and the output was a program generating sentences of the grammar. The TCG was implemented as a C language processor and used to produce large tests stressing certain language features [14].

- **1989, Wichmann and Davies:** A test suite was given for testing the syntax and semantics of Pascal. The test generator PPG (Pascal Program generator) was implemented in Pascal and produces Pascal validation suites. It generates self-checking correct programs on the basis of parameters given as input. A machine independent pseudo-random number generator was used to help in getting some degree of

repeatability. PPG works in a host target environment. The host would run PPG and the results were transferred to target either before or after completion depending upon the type of compiler used [31].

- **1990, Maurer:** Discussed data generator generators, in particular a generator based on the DGL (data generation language) [22] [23]. Its takes an enhanced test grammar as the input and generates tests according to the grammar. The test cases are generated by using the starting symbol of the input grammar. When a test production is selected, it is scanned from left to right and all non-terminals are replaced by data. Alternatives are selected at random. Rules can be specified to choose alternatives and to assign weights to them. A successful application of the method to compiler testing depends on data structures of the language. The method is easier to apply to C, while not so other languages.

- **1991, Denney:** A meta-interpreter was designed for Prolog using the language itself. There was a number of problems working with Prolog specification such as recursion, evaluation of predicates, etc.; the interpreter handled all such problems using a deterministic automaton. The meta-interpreter defines paths through the specification to be used as test cases. The specification automaton was generated dynamically as it executes the specification by translating Prolog's goal reduction states to the automaton states. Equivalence classes provide a basis for good test coverage and avoid wasting time on tests that are essentially equivalent. As test cases are being generated, the interpreter checks to see whether their equivalence class has already been used [5].

- **1992, Liyuan and Guangjun:** In their work, they used Purdom's algorithm to generate test cases automatically. They modified it to work with a

grammar for the programming language Jovial, and to produce Jovial programs that can be used to test Jovial compiler. A series of functional modules is set up to solve the context sensitive problem in program generating processes. The program generated are guaranteed to be short in length. The grammar can modify itself to generate programs with syntax error necessary for complete compiler testing [15].

- **1993, Kawata et al:** presented a test program generator TPGEN. It generates executable programs with self-checking code. Given a grammar, it generates a program by selecting production rules at random or in particular order specified by the user. When selecting the rules at random, the variables or functions defined in the declaration part will not coincide with those defined in execution portion. This problem is overcome by using a system function to store and retrieve information about declared variables [16].

Most of these methods are fully automated, but Celentano et al.'s method is only partially automated and the method by Wichmann and Jones is not automated at all. Generally, these methods considered real-life programming languages such as ADA, Fortran and Pascal. They mostly concentrates on the syntactic features and constructs of the respective programming languages. But none of them have clearly addressed the testing of the most critical, advanced features of modern languages. Also, most methods do not consider whether the compiler under test deals with the data declaration parts of the test programs properly. Therefore, these methods are more appropriate for simple syntax testing and they need to be extended to deal with complex semantics of languages [5].

Chapter 06

Purdom's Algorithm and Its Implementation

In the previous chapters we discussed various reasons and methods of compiler testing. The most suitable is functional testing (having a test case for each language feature) with structural coverage (each statement, branch or path in the program is to be traversed a least once). Thus, the most viable method of compiler testing so far is by generating a series of test cases (source programs), which are hand-written and for each of them the correct behavior of the compiler is verified.

For a parser, its flowchart is the same as the structure of the syntax chart of the language – traversing all branches in the syntax chart is equivalent to traversing all branches of the syntax analyzer. In addition, all table-driven parsers (e.g. bottom-up parsers) have a fixed control structure and all the information regarding the parsing of the language is stored in the table, which are thus the real target of testing. For recursive-descent parsers, the situation is much closer to the usual software testing and the the parser must be tested for both, the parsing and the processing. Test preparation effort depends on the language size and the size of tables, and increases as their sizes increases. The parser has to be available in source form to perform structural testing.

An important aspect of compiler testing is that there are several distinct levels of correctness to be considered.

- Lexical Correctness
- Syntactical /Context-Free Correctness
- Compile-time Correctness
- Run-Time Correctness
- Logical Correctness

For the first three levels of correctness we have to generate test cases, while the other two can be done during compiler verification.

It is important to generate test cases for the first three levels as:

- Lexically correct programs are used to verify the syntax related diagnostics.
- Syntactically correct program exercise the diagnostics concerning the correspondence between declaration and use of variables.
- Compile-time correct programs verify the correctness of behavior of parser and code generators.

The method for generating test data should fulfill the following objectives:

- The method should depend on the source language, yet should be independent of the compiler. It should also produce incorrect programs as well.
- The method should produce a set of test programs meeting some completeness criteria, rather than a randomly selected set [6].

Generating sentences is the first step in testing. These sentences are to be correct with respect to the given context-free grammar, but may be

incompatible with other features of the language such as variable declaration, use of identifiers etc. Execution of these test cases will verify the syntax analyzer (parser) of the compiler and also help test the diagnostic capabilities of the compiler to some extent by executing the contextually incompatible programs.

6.1 Purdom's Algorithm

Purdom in 1972 proposed a method for testing parsers automatically by generating test programs on the basis of the grammar with the objective of using each grammar rule at least once. The additional objective was to generate the sentences efficiently and as short as possible. According to him, a set of sentences using all the grammar rules is a good candidate for exercising most of the parser code or tables. As all programming languages are context sensitive, this method only confirms the syntactical aspect and there is no guarantee that these programs will execute correctly. Purdom's algorithm focuses on verifying the parser's correctness, not interested in checking the efficiency, performance, and other aspects [29].

Purdom's algorithm takes a context free grammar as input. It starts the generating process with the starting symbol ***S*** of the grammar. It keeps rewriting the non-terminals with RHS of matching rules and until all the non-terminals are eliminated and replaced by terminals. The algorithm follows the same pattern as a parsing algorithm with the difference that instead of recognizing tokens in the input string, it generates tokens for the output. This can be seen in the following high-logic description of the algorithm.

1. Push the ***S*** on the stack.
2. While the stack is non empty, repeat:
   - Pop the topmost element

- If this element is a terminal, match it with the input.
- If it is a non-terminal, choose a rule for the non-terminal, and push the symbols from the RHS of the rule onto the stack, in reverse order [19].

6.2 Power & Malloy's reformulation of Purdom's Algorithm

The original Purdom's algorithm presented in 1972 was described in a very imperative style and was very difficult to understand and implement. Power and Malloy reformulated the whole process and presented it in a very structured manner in three distinct phases . We have implemented their reformulation with an important modification of the third phase [18] [19].

6.3 Our implementation of Purdom's Algorithm

We used Purdom's algorithm to test the various LL(1) grammars for MACS and to generate test cases for MACS parsers. The final LL(1) grammar for MACS is given in appendix A. The grammar has 77 terminals, 90 non-terminals, and 301 productions.

Since it was anticipated that we would be working with various MACS grammars, it was desirable to make the grammar as input to the algorithm. Thus both our implementations, in Java and C++ follow the same architecture. The grammar is first input as three separate ASCII text files – the first (`terms.asc`) listing all terminals, the second (`nonterms.asc`) listing all terminals, and the third (`rules.asc`) listing all productions. This seemed more flexible as no specific notation distinguishing terminals from non-terminals is needed (though all MACS grammars we worked with adhered to the standard that terminals were names in upper cases). These three files are input into the first component, **Grammar** that processes the input files into a proper Java (or C++) class. The program **Grammar** thus generates Java or C++ code to be used by the

implementation of Purdom's algorithm, the program **Purdom**. This simplified the design and coding of **Purdom**, as we could work with a nice and rich class describing the input grammar. Every time the input grammar changed, all we needed was to run **Grammar** with the new grammar, generate a new grammar class and recompile **Purdom** with the new class.

Purdom is then executed and it generates MACS terminal sequences (sentences). These are manually transformed to syntactically correct MACS programs by replacing tokens with lexemes, they are the test data for MACS parser. The whole process is schematically depicted in Figure 6.1.

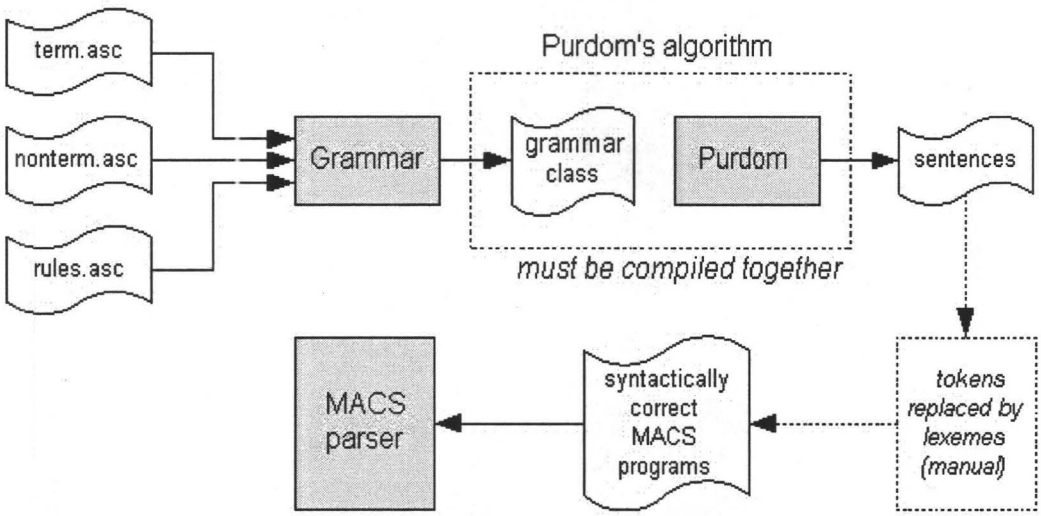

Figure 6.1 Working of our implementation of Purdom's algorithm

1. `terms.asc` contains the list of all the terminals in the language, one symbol per line (i.e. the newline is used as a separating character between them).
2. `nonterms.asc` contains the non- terminals of the language, also

one symbol per line.

3. `rules.asc` is the list of all production rules. The LHS symbol of the very first rule listed is assumed to be the starting non-terminal ***S***. Rules are in the following format: LHS non-terminal, followed by colon (:), followed by terminals and non-terminals of the RHS separated by at least one white space. The rule is terminated by semicolon. (;). An epsilon rule has : followed by ;
   The regular expression describing a rule
   <terminal> ' : ' (<terminal> | <non-terminal>) * ' ; '

The algorithm (**Purdom**) is divided into three phases with five subroutines. All the array names used for holding the intermediate results are kept unchanged through all three phases (e.g. the array called SLEN in all routines of phase I is the same array SLEN used in all routines of phase II). The division of the whole algorithm into three separate phases along the ideas of Power & Malloy makes it easier to describe, understand, and analyze the algorithm. The arrays with the intermediate result thus function as the connection (coupling) between the phases. However, the division into three separate phases has some negative impact on the execution speed.

In the following sections we describe the high-level logic of the phases.

6.3.1 Phase I (Shortest Terminal String)

In the first phase of Purdom's algorithm, three arrays SLEN, RLEN, and SHORT are computed.

- **SLEN:** is an array containing entries for all symbols of the grammar (i.e. terminals and non-terminals). For each non-terminal, the array is initialized

with ∞ (implemented as a maximal integer value), while for each terminal it is initialized with 1 (it will remain unchanged).

(*In the original description of the algorithm by Purdom, the length of the corresponding lexeme was to be used, we opted for this compromise, not knowing the actual lengths of lexemes. For instance, for the terminal* COMMA *we know that the length of the lexeme* ',' *will always be 1. On the other hand, the lexemes for* ID *can be of any length and so we would not know how to assign the lengths to the terminal* ID.)

The algorithm starts rewriting non-terminals using the production rules to discover the shortest length of derivation. At the end of this phase we will have the shortest length of string for each grammar symbol.

- **SLEN:** No. of steps in the derivation + Number of characters in the string (All other lengths in the algorithm are also calculated using this formula).

- **RLEN:** Array containing entries for each rule, each entry gives the length of the shortest string which we get using that rule. Again the length is the sum of the steps taken in the derivation and the number of terminals in the resulting string.

- **SHORT:** For each non-terminal we maintain an array such that it contains the production number, which gives us the shortest string.

We can check some aspects of the grammar being used by the end of phase I: if after the phase I is completed, any entry of SLEN is ∞ or if any entry of SHORT contains -1, it is an indication that the grammar is ambiguous or incomplete. There are some productions that are never used for deriving

(terminal) strings or some non-terminals have no associated rules. It is one of the unique methods to detect these errors in a context-free grammar.

6.3.2 Phase II (Shortest Derivation Algorithm)

Second phase uses the SLEN and RLEN computed in the previous phase and produces two new arrays, DLAN and PREV, to be used by the final phase.

- **DLEN:** for each non-terminal, it gives the length of the shortest string used it in its derivation.

- **PREV:** contains the rule number to be used to introduce a non-terminal in the shortest string derivation. Values are calculated for all non-terminals except for the starting symbol.

For the starting symbol ***S*** the PREV should be set to –1, as it cannot be introduced by any rule, all other entries of DLEN should be same as SLEN. At the end of this phase, DLEN should not be ∞ for any non-terminal and PREV should not be equal to –1, except for the starting symbol ***S***. If this happens, the grammar is incorrect.

6.3.3 Phase III (Sentence Generation Algorithm)

In the third phase, the sentences for the given language are generated. First, the starting symbol is pushed on the stack, and then as long as the stack contains any elements, the algorithm keeps popping the top of the stack and rewriting it using a production rule.

6.3.3.1 Choose a Rule

One of the goals of the algorithm is to use all the production rules at least once. The reformulation given by Brain and Malloy [18] [19] uses the production rules in sequence, which gives the maximum length strings.

Our modification uses the rules on the basis of the values in PREV and SHORT; whenever the algorithm finds a rule with a low value of PREV or SHORT, it replaces the existing one with it giving at the end the minimum length sentences.

For a non-terminal **A**, if a rule $A \rightarrow \alpha$ that has not yet been used exists, then it is chosen. If more than one such rule exist, then the one with the lowest values of PREV and SHORT is chosen. Otherwise, if a derivation $A \rightarrow \alpha \rightarrow \gamma_1 B \gamma_2$ exists such that **B** is a non-terminal not on the stack and a rule $B \rightarrow \beta$ exists that had not been used, then it uses the production $A \rightarrow \alpha$, which will then be rewritten using any of the α rules.

For each non-terminal on the stack, the algorithm maintains the following arrays:

- **ONST:** contains the occurrences of non-terminals on the stack. At the end it should be zero, their should not be any unused symbol on the stack.

- **MARK:** for each rule it contains either **true** or **false**, but at end of this phase all the entries should be **true**, which shows that each and every rule had been used at least once, the main requirement of Purdom's algorithm.

- **ONCE:** for each non-terminal it contains one of the following values:
  READY: the production number previously in ONCE has been used and the next time this non-terminal will be rewritten using a different production.
  UNSURE: the value of ONCE calculated in the last loop is not certain -- for some non-terminals it is the production number used to introduce that symbol in shortest string derivation and for some non-terminals this is not true.
  FINISHED: the non-terminal has been rewritten using all possible productions and cannot be used in any other way.
  INTEGER: contains the number of the production used to rewrite the symbol in some useful derivation.

In our implementation, we used an integer array for ONCE where the following values are represented as:

Ready	-1
Unsure	-2
Finished	-3
Integer	from 0 to onwards are the rules numbers.

6.4 Some of the sentences Generated By Our Implementation of Purdom's Algorithm

1. *Void A.a(A a1, const A b);*
2. *void A.a(bool b);*
3. *public void A.a,A.b;*
4. *class A;*
5. *class a{}*
6. *class a extends A;*
7. *class a{ }*

8. *A.B( );*
9. *main(const string [] a){ }*
10. *public shared void A.a;*
11. *public const void A.a;*
12. *private void A.a;*
13. *private shared void A.a;*
14. *private const void A.d;*
15. *shared void A.a;*
16. *shared public void A.a;*
17. *shared private void A.a;*
18. *shared const void A.a;*
19. *const void A.a;*
20. *const public void A.a;*
21. *const private void A.a;*
22. *const shared void A.a;*
23. *main({ })*

Purdom's generated Sentences	MACS Test Cases	Syntactically Correct	Semantically Correct
VOID CLASSNAME_DOT ID_LP CLASSNAME ID COMMA CONST CLASSNAME ID RP SEMICOL	void A.a(A a1, const A b);	X	X
VOID CLASSNAME_DOT ID_LP BOOL ID RP SEMICOL	void A.a(bool b);	X	X
PUBLIC VOID CLASSNAME_DOT ID COMMA CLASSNAME_DOT ID SEMICOL	public void A.a,A.b;		
CLASS CLASSNAME SEMICOL	class A;	X	X
CLASS ID LB RB	class a{}	X	X
CLASS ID EXTENDS CLASSNAME SEMICOL	class a extends A;	X	X
CLASSNAME_DOT CLASSNAME_LP RP SEMICOL	A.B( );	X	X
MAIN_LP CONST STRING LS RS ID RP LB RB	main(const string [] a){ }	X	X
PUBLIC SHARED VOID CLASSNAME_DOT ID SEMICOL	public shared void A.a;	X	
PUBLIC CONST VOID CLASSNAME_DOT ID SEMICOL	public const void A.a;	X	
PRIVATE VOID CLASSNAME_DOT ID SEMICOL	private void A.a;	X	
PRIVATE SHARED VOID CLASSNAME_DOT ID SEMICOL	private shared void A.a;	X	

PRIVATE CONST VOID CLASSNAME_DOT ID SEMICOL	private const void A.d;	X	
SHARED VOID CLASSNAME_DOT ID SEMICOL	shared void A.a;	X	
SHARED PUBLIC VOID CLASSNAME_DOT ID SEMICOL	shared public void A.a;	X	
SHARED PRIVATE VOID CLASSNAME_DOT ID SEMICOL	shared private void A.a;	X	
SHARED CONST VOID CLASSNAME_DOT ID SEMICOL	shared const void A.a;	X	
CONST VOID CLASSNAME_DOT ID SEMICOL	const void A.a;	X	
CONST PUBLIC VOID CLASSNAME_DOT ID SEMICOL	const public void A.a;	X	
CONST PRIVATE VOID CLASSNAME_DOT ID SEMICOL	const private void A.a;	X	
CONST SHARED VOID CLASSNAME_DOT ID SEMICOL	const shared void A.a;	X	
MAIN_LP LB RB	main({ })	X	

Table 6.1 Sentences Generated By the Algorithm

The psuedocode of the individual phases is given Appendix D.

Chapter 07

Conclusion

We implemented in Java and C++ Purdom's algorithm that generates simple sentences of a given LL(1) grammar. The algorithm is fast and generates the shortest sentences with the objective of using each production rule at least once. The algorithms was used to verify completeness and unambiguity of various LL(1) grammars for the programming language MACS, and to generate test data for a JavaCC-based top-down MACS parser.

Though Purdom's algorithm is well-referenced in many texts of compiler testing, there are very few implementations mentioned in literature. When they are, no implementation details are provided. The closest come the work by Power and Malloy, however their implementation of the third phase did not seem to work. We thus modified the third phase significantly, producing an algorithm generating usable test data.

We have achieved the same results with both the Java and the C++ implementations of Purdom's algorithm. The sentences generated are mostly syntactically and semantically correct with a few exceptions which are semantically incorrect (see Table 6.1).

The test generator only automates the input part of testing, leaving it to the humans to check the correctness of the object code generated by the compiler. The ideal would be to produce test programs together with the expected trace of the results of their execution, which is a very difficult problem.

Appendix A List of Terminals of the final LL(1) grammar for MACS Language

AND, ASSIG, BOOL, BREAK, CATCH, CHAR, CHAR_LIT, CLASS, CLASSNAME, CLASSNAME_DOT, CLASSNAME_LP, CLASSNAME_RP, COLON, COMMA, CONST, CONTINUE, DOT, ELSE, EQ, EXTENDS, FALSE, FLOAT, FLOAT_LIT, FOR, GE, GOTO, GT, GTGT, ID, IDOF, ID_COLON, ID_LP, IF, INT, INT_LIT, LB, LE, LP, LS, LT, LTLT, MAIN_LP, MINUS, MINUSMINUS, MOD, NEQ, NOREF, NOT, OR, PARENT_DOT, PARENT_LP, PASSIG, PEEKNOTELSE, PERM, PLUS, PLUSPLUS, PRETURN, PRIVATE, PUBLIC, RB, RETURN, RP, RS, SEMICOL, SHARED, SIZEOF, SLASH, STAR, STRING, STRING_LIT, TERMINATE, THROW, TRUE, TRY, TYPEOF, VOID, WHILE

Appendix B List of Non-Terminals of the final LL(1) grammar for MACS Language

ArgType, ArgType1, Args, Args1, ArrayDim, AttrDecl, AttrDef, AttrMethodDecl, AttrMethodDecl1, AttrMethodDecl2, AttrMethodDef, Block, Block1, CastOrPexp, Catch, Catch1, ClassBody, ClassDeclDef, ClassDeclDef1, ClassDeclDef2, ClassMember, Cond, Cond1, ConstrDecl, ConstrDef, DataType, Epilog, Epilog1, Expr, Factor, Factor1, ForStm, IfStm, IfStm1, Init, MainSection, MethodBody, Params, Params1, ParentConstr, PassingSpec, Prefix, Prefix10, Prefix13, Prefix14, Prefix20, Prefix23, Prefix24, Prefix30, Prefix31, Prefix32, Prefix34, Prefix40, Prefix41, Prefix42, Prefix43, Program, Program1, Prolog, Prolog1, Ref, Ref1, Ref2, Ref3, Ref4, Ref5, Ref6, ReturnStm, SimpleExpr, SimpleExpr1, Stm, StringLit, StringLit1, StringLit2, Term, Term1, ThenBlock, ThenBlock1, Ustm, VarDef1, VarDef11, VarDef111, VarDef12, VarDef2, VarDef21, VarDef211, VarDef22, VarDef3, VarDef4, WhileStm

Appendix C The final LL(1) Grammar for MACS language

Program: MAIN_LP MainSection Program1;
Program: CLASSNAME_DOT ConstrDecl Program1;
Program: CLASS ClassDeclDef Program1;
Program: PUBLIC Prefix10 AttrMethodDecl Program1;
Program: PRIVATE Prefix20 AttrMethodDecl Program1;
Program: SHARED Prefix30 AttrMethodDecl Program1;
Program: CONST Prefix40 AttrMethodDecl Program1;
Program: BOOL ArrayDim AttrMethodDecl1 Program1;
Program: CHAR ArrayDim AttrMethodDecl1 Program1;
Program: FLOAT ArrayDim AttrMethodDecl1 Program1;
Program: INT ArrayDim AttrMethodDecl1 Program1;
Program: STRING ArrayDim AttrMethodDecl1 Program1;
Program: CLASSNAME ArrayDim AttrMethodDecl1 Program1;
Program: VOID CLASSNAME_DOT ID_LP Args SEMICOL Program1;
Program1: MAIN_LP MainSection Program1;
Program1: CLASSNAME_DOT ConstrDecl Program1;
Program1: CLASS ClassDeclDef Program1;
Program1: PUBLIC Prefix10 AttrMethodDecl Program1;
Program1: PRIVATE Prefix20 AttrMethodDecl Program1;
Program1: SHARED Prefix30 AttrMethodDecl Program1;
Program1: CONST Prefix40 AttrMethodDecl Program1;
Program1: BOOL ArrayDim AttrMethodDecl1 Program1;
Program1: CHAR ArrayDim AttrMethodDecl1 Program1;
Program1: FLOAT ArrayDim AttrMethodDecl1 Program1;
Program1: INT ArrayDim AttrMethodDecl1 Program1;
Program1: STRING ArrayDim AttrMethodDecl1 Program1;
Program1: CLASSNAME ArrayDim AttrMethodDecl1 Program1;
Program1: VOID CLASSNAME_DOT ID_LP Args SEMICOL Program1;
Program1:;
MainSection: LB MethodBody;
MainSection: CONST STRING LS RS ID RP LB MethodBody;
AttrMethodDecl: DataType AttrMethodDecl1;

```
AttrMethodDecl1: CLASSNAME_DOT AttrMethodDecl2;
AttrMethodDecl2: ID_LP Args SEMICOL;
AttrMethodDecl2: ID AttrDecl;
AttrDecl: SEMICOL;
AttrDecl: COMMA CLASSNAME_DOT ID AttrDecl;
ConstrDecl: CLASSNAME_LP Args SEMICOL;
ClassDeclDef: ID ClassDeclDef1;
ClassDeclDef: CLASSNAME ClassDeclDef1;
ClassDeclDef1: EXTENDS CLASSNAME ClassDeclDef2;
ClassDeclDef1: LB ClassBody;
ClassDeclDef1: SEMICOL;
ClassDeclDef2: LB ClassBody;
ClassDeclDef2: SEMICOL;
ClassBody: ClassMember ClassBody;
ClassBody: RB;
ClassMember: CLASSNAME_LP ConstrDef;
ClassMember: Prefix DataType AttrMethodDef;
DataType: BOOL ArrayDim;
DataType: CHAR ArrayDim;
DataType: FLOAT ArrayDim;
DataType: INT ArrayDim;
DataType: STRING ArrayDim;
DataType: CLASSNAME ArrayDim;
DataType: VOID;
ArrayDim: LS RS ArrayDim;
ArrayDim:;
Prefix: PUBLIC Prefix10;
Prefix: PRIVATE Prefix20;
Prefix: SHARED Prefix30;
Prefix: CONST Prefix40;
Prefix:;
Prefix10: SHARED Prefix13;
Prefix10: CONST Prefix14;
Prefix10:;
Prefix13: CONST;
```

```
Prefix13:;
Prefix14: SHARED;
Prefix14:;
Prefix20: SHARED Prefix23;
Prefix20: CONST Prefix24;
Prefix20:;
Prefix23: CONST;
Prefix23:;
Prefix24: SHARED;
Prefix24:;
Prefix30: PUBLIC Prefix31;
Prefix30: PRIVATE Prefix32;
Prefix30: CONST Prefix34;
Prefix30:;
Prefix31: CONST;
Prefix31:;
Prefix32: CONST;
Prefix32:;
Prefix34: PUBLIC;
Prefix34: PRIVATE;
Prefix34:;
Prefix40: PUBLIC Prefix41;
Prefix40: PRIVATE Prefix42;
Prefix40: SHARED Prefix43;
Prefix40:;
Prefix41: SHARED;
Prefix41:;
Prefix42: SHARED;
Prefix42:;
Prefix43: PUBLIC;
Prefix43: PRIVATE;
Prefix43:;
AttrMethodDef: ID Init AttrDef;
AttrMethodDef: ID_LP Args LB MethodBody;
AttrDef: COMMA ID Init AttrDef;
```

AttrDef: SEMICOL;
ConstrDef: Args LB ParentConstr MethodBody;
ParentConstr: PARENT_LP Params SEMICOL;
ParentConstr:;
PassingSpec: AND;
PassingSpec:;
ArgType: CONST ArgType1 PassingSpec;
ArgType: BOOL ArrayDim PassingSpec;
ArgType: CHAR ArrayDim PassingSpec;
ArgType: FLOAT ArrayDim PassingSpec;
ArgType: INT ArrayDim PassingSpec;
ArgType: STRING ArrayDim PassingSpec;
ArgType: CLASSNAME ArrayDim PassingSpec;
ArgType1: BOOL ArrayDim;
ArgType1: CHAR ArrayDim;
ArgType1: FLOAT ArrayDim;
ArgType1: INT ArrayDim;
ArgType1: STRING ArrayDim;
ArgType1: CLASSNAME ArrayDim;
Args: RP;
Args: ArgType ID Args1;
Args1: RP;
Args1: COMMA ArgType ID Args1;
VarDef1: CONST VarDef11;
VarDef1: BOOL ArrayDim VarDef12;
VarDef1: CHAR ArrayDim VarDef12;
VarDef1: FLOAT ArrayDim VarDef12;
VarDef1: INT ArrayDim VarDef12;
VarDef1: STRING ArrayDim VarDef12;
VarDef1: CLASSNAME ArrayDim VarDef12;
VarDef2: PERM VarDef21;
VarDef2: BOOL ArrayDim VarDef22;
VarDef2: CHAR ArrayDim VarDef22;
VarDef2: FLOAT ArrayDim VarDef22;
VarDef2: INT ArrayDim VarDef22;

VarDef2: STRING ArrayDim VarDef22;
VarDef2: CLASSNAME ArrayDim VarDef22;
VarDef3: ID Init VarDef4;
VarDef4: COMMA ID Init VarDef4;
VarDef4:;
VarDef11: BOOL ArrayDim VarDef111;
VarDef11: CHAR ArrayDim VarDef111;
VarDef11: FLOAT ArrayDim VarDef111;
VarDef11: INT ArrayDim VarDef111;
VarDef11: STRING ArrayDim VarDef111;
VarDef11: CLASSNAME ArrayDim VarDef111;
VarDef21: BOOL ArrayDim VarDef211;
VarDef21: CHAR ArrayDim VarDef211;
VarDef21: FLOAT ArrayDim VarDef211;
VarDef21: INT ArrayDim VarDef211;
VarDef21: STRING ArrayDim VarDef211;
VarDef21: CLASSNAME ArrayDim VarDef211;
VarDef12: ID Init VarDef4;
VarDef22: ID Init VarDef4;
VarDef111: ID Init VarDef4;
VarDef211: ID Init VarDef4;
Init: ASSIG Expr;
Init: PASSIG Expr;
Init:;
MethodBody: ID_COLON UStm MethodBody;
MethodBody: SEMICOL MethodBody;
MethodBody: IF IfStm MethodBody;
MethodBody: FOR ForStm MethodBody;
MethodBody: WHILE WhileStm MethodBody;
MethodBody: GOTO ID SEMICOL MethodBody;
MethodBody: CONTINUE SEMICOL MethodBody;
MethodBody: BREAK SEMICOL MethodBody;
MethodBody: TERMINATE SEMICOL MethodBody;
MethodBody: RETURN ReturnStm MethodBody;
MethodBody: PRETURN Expr SEMICOL MethodBody;

MethodBody: THROW Expr SEMICOL MethodBody;
MethodBody: TRY LB MethodBody CATCH LP Catch MethodBody;
MethodBody: PERM VarDef1 MethodBody;
MethodBody: CONST VarDef2 MethodBody;
MethodBody: BOOL ArrayDim VarDef3 MethodBody;
MethodBody: CHAR ArrayDim VarDef3 MethodBody;
MethodBody: FLOAT ArrayDim VarDef3 MethodBody;
MethodBody: INT ArrayDim VarDef3 MethodBody;
MethodBody: STRING ArrayDim VarDef3 MethodBody;
MethodBody: CLASSNAME ArrayDim VarDef3 MethodBody;
MethodBody: RB;
Catch: ID RP Catch1;
Catch1: LB MethodBody;
Stm: ID_COLON UStm;
Stm: UStm;
UStm: SEMICOL;
UStm: IF IfStm;
UStm: FOR ForStm;
UStm: WHILE WhileStm;
UStm: GOTO ID SEMICOL;
UStm: CONTINUE SEMICOL;
UStm: BREAK SEMICOL;
UStm: TERMINATE SEMICOL;
UStm: RETURN ReturnStm;
UStm: PRETURN Expr SEMICOL;
UStm: THROW Expr SEMICOL;
UStm: TRY LB MethodBody CATCH LP Catch;
UStm: Expr SEMICOL;
IfStm: LP Cond ThenBlock IfStm1;
IfStm1: ELSE Block;
IfStm1: PEEKNOTELSE;
ThenBlock: LB ThenBlock1;
ThenBlock: UStm;
ThenBlock1: RB;
ThenBlock1: Stm ThenBlock1;

Block: LB Block1;
Block: UStm;
Block1: RB;
Block1: Stm Block1;
ForStm: LP Prolog Cond1 Epilog Block;
Cond1: SEMICOL;
Cond1: Expr SEMICOL;
Prolog: SEMICOL;
Prolog: Expr Prolog1;
Prolog1: SEMICOL;
Prolog1: COMMA Expr Prolog1;
Epilog: RP;
Epilog: Expr Epilog1;
Epilog1: RP;
Epilog1: COMMA Expr Epilog1;
Cond: RP;
Cond: Expr RP;
WhileStm: LP Cond Block;
ReturnStm: SEMICOL;
ReturnStm: Expr SEMICOL;
Expr: Factor;
Factor: Term Factor1;
Factor1: PLUS SimpleExpr Factor1;
Factor1: MINUS SimpleExpr Factor1;
Factor1: LT SimpleExpr Factor1;
Factor1: LE SimpleExpr Factor1;
Factor1: LTLT SimpleExpr Factor1;
Factor1: GT SimpleExpr Factor1;
Factor1: GE SimpleExpr Factor1;
Factor1: GTGT SimpleExpr Factor1;
Factor1: AND SimpleExpr Factor1;
Factor1: OR SimpleExpr Factor1;
Factor1: EQ SimpleExpr Factor1;
Factor1: NEQ SimpleExpr Factor1;
Factor1: MOD SimpleExpr Factor1;

```
Factor1: ASSIG Expr;
Factor1: PASSIG Expr;
Factor1:;
Term: SimpleExpr Term1;
Term1: STAR SimpleExpr Term1;
Term1: SLASH SimpleExpr Term1;
Term1:;
SimpleExpr: LP CastOrPexp;
SimpleExpr: PLUS SimpleExpr1;
SimpleExpr: MINUS SimpleExpr1;
SimpleExpr: NOT SimpleExpr1;
SimpleExpr: SIZEOF SimpleExpr1;
SimpleExpr: TYPEOF SimpleExpr1;
SimpleExpr: IDOF SimpleExpr1;
SimpleExpr: PLUSPLUS SimpleExpr1;
SimpleExpr: MINUSMINUS SimpleExpr1;
SimpleExpr: Ref;
SimpleExpr1: FALSE;
SimpleExpr1: TRUE;
SimpleExpr1: NOREF;
SimpleExpr1: CHAR_LIT;
SimpleExpr1: FLOAT_LIT;
SimpleExpr1: INT_LIT;
SimpleExpr1: STRING_LIT StringLit;
SimpleExpr1: Ref;
StringLit: LS Expr StringLit1;
StringLit:;
StringLit1: RS;
StringLit1: COLON Expr RS StringLit2;
StringLit2: LS Expr RS;
StringLit2:;
CastOrPexp: BOOL RP SimpleExpr1;
CastOrPexp: CHAR RP SimpleExpr1;
CastOrPexp: FLOAT RP SimpleExpr1;
CastOrPexp: INT RP SimpleExpr1;
```

```
CastOrPexp: STRING RP SimpleExpr1;
CastOrPexp: CLASSNAME_RP SimpleExpr1;
CastOrPexp: Expr RP Ref3;
Ref: CLASSNAME_DOT Ref1;
Ref: PARENT_DOT Ref1;
Ref: Ref2;
Ref1: PARENT_DOT Ref1;
Ref1: Ref2;
Ref2: ID Ref3;
Ref2: ID_LP Params Ref3;
Ref3: DOT Ref2;
Ref3: LS Ref4;
Ref3: PLUSPLUS;
Ref3: MINUSMINUS;
Ref3:;
Ref4: RS Ref3;
Ref4: Expr Ref5;
Ref5: RS Ref3;
Ref5: COLON Expr RS Ref6;
Ref6: LS Expr RS;
Ref6:;
Params: RP;
Params: Expr Params1
Params1: RP;
Params1: COMMA Expr Params1;
```

Appendix D The pseudo code of the three phases of our implementation of Purdom's algorithm

Phase One "Shortest Terminal String"

```
Input:  List of symbols, List of  rules
Output: SLEN, RLEN, SHORT

Void Init()
{
for ( each symbol s)
{
if (s is a  terminal)
        SLEN [s] = 1;
else
        {
                SLEN [s] = INIT_MAX ;
                SHORT [s] = -1;
        } // if
} // for
 for (each rule r )
{
        RLEN[ r] = INIT_MAX ;
} // for
} // Init( )
Void ShortestTerminalString( )
{
boolean One = true ;
```

```
while( One ) {
        One = false;
        for( each rule r ) {
            int sumOne = 1;boolean big = false;
            for ( each symbol s at the RHS of the production rule ){
                if ( this is a null production )
                    break;
                if ( SLEN [s] )  = = INT_MAX)
                    {
                        big = true;
                        break;
                    } //if
                    else{
                        sumOne + = SLEN [s];
                      } //else
                    } //for
                if ( !big and  sumOne < RLEN [r] ){
                    RLEN[r] = sumOne;
                    if (sumOne < SLEN [LHS[r]] {
                                SHORT [LHS[r]] = r;
                                SLEN [LHS[r]] = sumOne;
                                One = true;
                } //if
                } //if
            } //for
        } //while
} // ShortestTerminalString( )
```

Phase Two "Shortest Derivation String"

```
Input: SLEN, RLEN
Output: DLEN PREV

void Init( ){
for (each symbol s)
{
        if (s is a non-terminal)
                {
                DLEN[s] = INIT_MAX;
                PREV [r ] = -1;
                } // if
} // for
} // Init ( )

void ShortestDerivationString( ) {
                boolean two;
                two = true;
                DLEN [strarting symbol S]=SLEN [LHS[first rule r0] ];
                while( two )
                {
                        two = false;
                        for( each rule r )
                        {
                                if ( RLEN[r] = = INT_MAX )
                                {
```

```
        continue;
} //if

if ( DLEN[LHS[r] ] = = INT_MAX)
{
        continue;
} //if

if ( SLEN [LHS [r] ] = = INT_MAX )
{
        continue;
} //if
int sumtwo;
sumtwo = Dlen [ LHS[r] ] + RLEN [r] – SLEN [LHS[r] ];
for (each symbol s at the RHS of r)
{
        if (s in a non-terminal ){
                if ( sumtwo < DLEN [s])
                {
                        twochange = true ;

                        DLEN [s] = sumtwo ;

                        PREV [s] =r ;

                }//if
        }//if
}//for
```

```
                    }//for

                }//while
    }//ShortestDerivationString( )
```

Phase Three "Generate Sentence"

Input: PREV,SHORT

Output:Sentences

Auxiliary Functions : calcshort, load_once, process_Stack

```
void Init( ){
for (each symbol s)
{
if (s is a non-terminal )
{
        if (  s is the starting symbol)
        {
                ONCE[s] = _ready;
                ONST[s] = 1;
        } // if
        else
        {
                ONCE[s] = _ready;
                ONST[s] = 0;
        } //else
} //if
} // for

for( each rule r)
```

```
{
        MARK [ r ] = _true ;
} //for
} // Init( )

int calcshort ( non-terminal n )
{
int prodno;
prodno = SHORT [n];
MARK [prodno] =_true;
if ( ONCE[n] ! =_finished)
{
        ONCE[n] =_ready;
}//if
return prodno;
}//calcshort( )

void load_once( )
{
for( all rules r )
{
        if ( MARK[r ] = =_false)
        {
                ONCE[LHS[r]] = rule number ;
                MARK[r] =_true;
        } //if
} //for
} //load_once( )
```

```
boolean process_Stack(int prodno,boolean moresentence)
{
ONST [ nt.ind ] - -;
        for ( each symbol s at the RHS of rule r in reverse order )
        {
                Stack.Push(s);
                        if (s is a non-terminal)
                                {
                                        ONST[s] ++ ;
                                } // if
        } // for
boolean done = false;
                                while( !done )
                                {
                                        if ( Stack.Empty ( ) )
                                        {
                                                moresentence = false;
                                                break ;
                                        } // if
                                        else
                                        {
                                                nt = Stack.Pop( );
                                                if (nt is a terminal )
                                                        Print this terminal
                                                else done = true;
                                        } // else
                                } // while ( !done)
return moresentence;
} // process_Stack()
```

```
void GenerateSentence()
{
boolean  done = false;
boolean moresentence;
while(!done)
{
        if ( ONCE[ Start ] = =_finished)
                {
                        break;
                } // if
        ONST [Start] =1;nt = Start;
        moreentence = true;
        while( moresentence )
                        {
                        int once_nt;
                        once_nt = ONCE[nt];
                        if( nt = = Start and once_nt = =_finished )
                        {
                                done = true;
                                break;
                        } // if
                        else if ( once_nt = =_finished )
                                prodno = calcshort (nt);
                        else if ( once_nt > = 0)
                        {
                                prodno = once_nt;
                                ONCE [nt] =_ready;
                        }//if
```

```
else
{
        load_once( );
        for( each symbol s )
        {

                if (  s ! = Start and ONCE[s] >= 0 )
                {
                        if ( s is a non-terminal )
                        {
                                k = PREV[s];
                                while(k>=0)
                                {
                                if ( ONCE [LHS[k]] > = 0)
                                        {
                                                break;
                                        }//if
                                        else
                                        {

                                        if ( ONST[s] ] = = 0)
                                        {

                                        ONCE [LHS[k] ]= k;

MARK[LHS[k]]=_true;

                                        } //if
                                else
                                ONCE [LHS[k] ]= _unsure;
```

```
                                        }//else
                                            k=PREV[ LHS[k]];
                                        }//while
                                    }//if
                            }//if
                            //for
                    for(each symbol s)
                    {
                            if(ONCE[s] = =_ready)
                                    {
                            if(s is a non-terminal)
                            {

                                        ONCE [s]=_finished;
                    }//if
                                    }//if
                            }//if
            }//for
                    if( nt = = Start )
                            break;
                    else if (ONCE [nt] < 0 )
                            prodno = calcshort( nt );
                    else if (ONCE [nt ] > = 0 )
                    {
                            prodno = ONCE[nt ];
                            ONCE [nt ]=_ready;
                    }
                    }//else
                    moresentence=process_Stack(prodno,moresentence);
```

```
            }//while(moresentence)
        }//while(!done)

}//GenerateSentence()
```

References

[1] A. W. Appel and M. Ginsburg, **Modern Compiler Implementation in C**, Cambridge university press, 2001

[2] F. Bazzicchi and I. Spadafora, **An automatic generator for compiler testing**, IEEE Trans. Soft. Eng., 8 (4) (1982), pp 343-353

[3] B. Beizer, **Software testing techniques**, 2nd ed., Van Nostrand Reinhold, New York, 1990

[4] A.S. Boujarwah and K. Saleh, **Compiler test case generation: a survey and assessment**, Information and software technology, Volume 39, Issue 9, March 1997, pp 617-625

[5] A. Boujarwah and K. Saleh, **Compiler test suite: evaluation and use in an automated environment**, Information and Software Technology 36, (10) 1994, pp 607-614

[6] A. Celentano, S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, G. Granata, F. Savoretti, **Compiler testing using a sentence generator**, Software-Practice and Experience, IO (June 1980) pp 897-918

[7] N. Chomsky, **Three models for the description of language**, IRE Transactions on Information Theory (2), 1956, pp 113–124

[8] N. Chomsky and M.P. Schutzenberger, **The algebraic theory of context-free languages**, In: *Computer Programming and Formal Systems*, North-Holland, Amsterdam, 1963, pp 118-161

[9] C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko, and K. Cline, **A Certifying Compiler for Java**, ACM SIGPLAN Notices, 2000

[10] J. Earley, **An Efficient Context–Free Parsing Algorithm**, Communications of the ACM, vol 13(2), 1970, pp 94-102

[11] F. Franek, **Compiler Design and Implementation using C++ and Java**, manuscript, April 2007 (to be published by Jones and Bartlett)

[12] J. B. Goodenough, **The Ada Compiler Validation Capability**, *Proceedings of the ACM-SIGPLAN symposium on The ADA programming language*, ACM 1980, pp 1-8.

[13] K.V. Hanford, **Automatic generation of test cases**, IBM System Journal, 1970, pp 242-258

[14] W. Homer and R. Schooler, **Independent testing of compiler phases using a test case generator,** *SOFTWARE-PRACTICE AND EXPERIENCE*, VOL. 19 (1), 1989, pp 53-62

[15] JIANG Liyuan, HUANG Guangjun. **An automatic system of generating test cases for compiler**. Journal of Northwestern Polytechnical University, 1992, 10 (2), pp153-158 (In Chinese)

[16] H. KAWATA, H. SAIJO, C. SHIOYA, **A practical test program generator based on attributed grammar,** Fujitsu scientific and technical journal 1993, vol. 29, no 2, pp 128-136

[17] J.S. Marr and P.K. Lawlis, **Automatic Determination of Recommended test Combinations For Ada Compilers**, *Proceedings of the eighth annual Washington Ada symposium & summer SIGAda meeting on Ada: software: foundation for competitiveness*, ACM, 1991, pp 77-89

[18] B.A. Malloy and J.F. Power; **An interpretation of Purdom's algorithm for automatic generation of test cases**. *Proceedings of 1st Annual International Conference on Computer and Information Science*, Orlando, Florida, USA, October 3-5, 2001

[19] B.A. Malloy and J.F. Power, **A Top-down Presentation of Purdom's Sentence Generation Algorithm**, Technical Report NUIM-CS-TR-2005-04, National University of Ireland, Maynooth, 2005

[20] B.A. Malloy and J. T. Waldron, **Applying Software Engineering Techniques to Parser Design: The Development of a C# Parser**, ACM International Conference Proceeding Series; Vol. 30, pp 75-82

[21] R. Mandl, **Orthogonal latin squares: an application of experiment**

design to compiler testing, Communications of the ACM, 28 (IO) (1985), pp 1054-1058

[22] P. Maurer, **Generating Test Data with Enhanced Context Free Grammars**, Department of Computer Science, Technical Report Number SE-2, Baylor University, Waco, TX, 76798, 1990.

[23] P. Maurer, **THE DESIGN AND IMPLEMENTATION OF A GRAMMAR-BASED DATA GENERATOR**, Department of Computer Science, Technical Report Number SE-1, Baylor University, Waco, TX, 76798, 1991

[24] G. Myers, **The Art of Software Testing**, Wiley, New York, 1979

[25] G.C. Necula and P. Lee, **The design and implementation of a certifying compiler**, ACM SIGPLAN Notices, Volume 39, Issue 4, 2004, pp 612-625

[26] P. Oliver, **Experiences in Building and Using Compiler Validation Systems**. In Merwin, R. and J. Zanca, editors, *4. FIPS Conference Proceedings*, New Jersey, AFIPS Press, June 1979, pp 1051-1057

[27] A. Pnueli, M. Siegel and E. Sigerma, **Translation Validation**, Proceedings of 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, pp 151-166

[28] J.F. Power and B.A. Malloy, **Metric–Based Analysis of Context Free Grammars**, *Proceedings of 8th International Workshop on Program Comprehension*, IEEE Computer Society: Los Alamitos, CA, 200, pp 171-178

[29] P. Purdom, **A Sentence Generator For Testing Parsers**, BIT, vol 12, April 1972, pp 366-375

[30] J. Riehl, **Grammar Based Unit Testing for Parsers**, Master's Thesis, University of Chicago, Dept. of Computer Science, 2004

[31] B. A. Wichmann and M. Davies, **Experience with a Compiler Testing Tool**, Technical report, National Physical Laboratory, England (NPL Report DITC 138/89), 1989

[32] N. Wirth, **Compiler Construction**, Addison Wesley 1996

[33] L.Zuck, A. Pnueli, and R. Leviathan, **Validation of Optimizing Compilers**, Technical Report MCS01-12, Weizmann Institute of Science and New York University, 2001

[34] L.Zuck, A. Pnueli, Y. Fang, and B. Goldberg, **VOC: A Translation Validator for Optimizing Compilers,** *Proceedings of International Workshop on Compiler Optimization Meets Compiler Verification* COCV 2002, Grenoble, France, April 13, 2002

[35] L.Zuck and A. Pnueli, Y. Fang, and B. Goldberg, **VOC: A Methodology for Translation Validation of Optimizing Compilers**, Journal of Universal Computer Science, pp. 223-247, 2003.

[36] http://www.cs.nyu.edu/validation/description.html

[37] ISO / IEC 14977: 1996 (E)