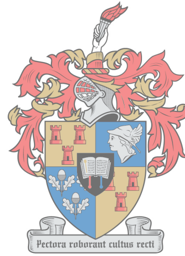
Test Case Generation for Context Free Grammars

by

M. H. Esterhuizen



UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

100
1918 · 2018

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science in the Faculty of Science at Stellenbosch University

Supervisor: Prof. Bernd Fischer

March 2018

The financial assistance of the Council for Scientific and Industrial Research (CSIR), through the Centre for Artificial Intelligence Research (CAIR), towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at are those of the author and are not necessarily to be attributed to the CSIR or CAIR.

Stellenbosch University https://scholar.sun.ac.za

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2018

Copyright © 2018 Stellenbosch University
All rights reserved.

Stellenbosch University https://scholar.sun.ac.za

Abstract

Test Case Generation for Context Free Grammars

Marvin Heinrich Esterhuizen

Division of Computer Science,
Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, 7602 Matieland, South Africa.

Thesis: MSc Computer Science

March 2018

Software testing, despite decades of ongoing research, still forms a significant part of the development cycle. When the input domain of a software system must satisfy structural constraints, such as those specified by file formats or command line arguments, test input construction has the potential to be an extremely time consuming process.

In this work we investigate how structured test inputs may be constructed through the use of formal grammars and how the input domain of a software system may be adequately represented by the constructed test inputs. We develop a framework which facilitates the automatic generation of test inputs and use it to implement a number of test input generation techniques to satisfy specific coverage criteria. We also introduce a method of test case case generation that exploits the structure of LR-automata to generate sets of passing and failing test cases.

The investigations conducted in this work focusses on primarily two areas. Firstly, we investigate how the LR-automata based method of generating test inputs compares to the existing methods. Secondly we seek to verify that the LR-automata and existing methods of test input generation conform to their expected running times.

The framework and algorithms are evaluated, in the context of two university level compiler courses as a method to assess parsers, generated from handwritten grammars, submitted by students and compared to the method of assessing submissions with test inputs constructed manually. We find that, in our sample set, assessment based on the positive test inputs generated by the LR-automata based method correlates most closely to assessment based on manually constructed test inputs. This indicates that the method would be most appropriate as a replacement for manually constructing test inputs. The results obtained from assessment based on negative test inputs generated by both, grammar an LR-automata, based methods is found to be unsatisfactory as there is no cor-

Stellenbosch University https://scholar.sun.ac.za

relation between them and the results obtained through assessment with a set of manually constructed negative test inputs.

An assessment of the performance of the algorithms is also given. We find that the running time of the existing methods of test input generation, based on context free grammars, varies linearly with respect to the size of the grammar and that the running time of the LR-automata based methods varies linearly with respect to the size of the constructed parsing automata.

Stellenbosch University https://scholar.sun.ac.za

Uittreksel

Test Case Generation for Context Free Grammars

Marvin Heinrich Esterhuizen

Verdeling van Rekenaar Wetenskap,
Department Wiskundige Wetenskappe,
Universiteit van Stellenbosch,
Privaatsak X1, 7602 Matieland, Suid Afrika

Tesis: MSc Rekernaar Wetenskap

Maart 2018

Sagteware toetsing, ten spyte van dekades van voortgesette navorsing, vorm steeds ‘n belangrike deel van die ontwikkelings siklus. Wanneer die insette domein van ‘n sagteware stelsel strukturele beperkings moet bevredig, soos wat gespesifiseer word deur lêerformate of bevellyn argumente, het toetsinsette konstruksie die potensiaal om ‘n uiters tydrowende proses te wees.

In hierdie werk ondersoek ons hoe gestruktureerde toetsinsette deur die gebruik van formele grammatikas kan gebou word en hoe die insetdomein van ‘n sagteware sisteem voldoende verteenwoordig word deur die konstruksie van toetsinsette. Ons ontwikkel ‘n raamwerk wat die outomatiese generering van toetsinsette fasiliteer en gebruik dit om ‘n aantal toetsinvoer genereringtegnieke te implementeer om spesifieke dekkingskriteria te bevredig. Ons stel ook ‘n metode van toetsgevalgenerering bekend wat die struktuur van ‘n LR-automaat gebruik om stelle postiewe en negatiewe toetsgevalle te genereer.

Die ondersoek wat in hierdie werk uitgevoer word fokus hoofsaaklik op twee gebiede. Eerstens, ondersoek ons hoe die LR-outomaat gebaseerde metode om toetsinsette te genereer vergelyk met die bestaande metodes. Tweedens poog ons om te verifeer dat die LR-automaat en bestaande metodes van toetsinvoergenerering ooreenstem met hul verwagte looptye.

Die raamwerk en algoritmes word geëvalueer, in die konteks van twee universiteits kompileerder kurses opdragte as ‘n metode om sintaksontleders te evalueer, wat gegenereer is vanaf handgeskrewe grammatikas, en word vergelyk met die metode om voorleggins met handgeskrewe toetsinsette te assesseer. Ons vind dat in ons steekproef, assessering gebaseer op positiewe toetsinsette wat gegenereer word deur die LR-automaat gebaseerde metode, die beste korreleer met assessering gebaseer op handopgestelde toetsinsette. Dit dui aan dat die metode mees geskik sal wees as ‘n plaasvervanger vir toetsing deur middel van handopgestelde toetsinsette. Die resultate verkry uit assessering gebaseer op negatiewe toetsinsette gegenereer deur grammatika en LR-outomaat, is gevind om onbevredigend te wees aangesien daar geen verband tussen hierdie metodes

Stellenbosch University https://scholar.sun.ac.za

en die resultate verkry deur assessering met ‘n stel handopgestelde negatiewe toetsinsette is nie.

‘n Beoordeling van die looptyd van die algoritmes word ook gegee. Ons vind dat die looptyd van die bestaande metodes van toetsinsette generasie, gebaseer op konteksvrye grammatikas, wissel lineêris met betrekking to die grootte van die grammatika en dat die looptyd van die LR-outomaat gebaseerder metodes wissel lineêris met betrekking tot die grootte van die gekonstrueerde outomaat.

Stellenbosch University https://scholar.sun.ac.za

Acknowledgements

I would like to express my sincere gratitude to the following people. To my supervisor, Bernd Fischer, who provided insight, guidance and inspiration throughout the writing of this dissertation. To my family who encouraged and supported me. To my friends who provided the much appreciated fleeting moments of relief from a hard days work and to Lynique Bell who, like me, was very lonely towards the completion of this dissertaion.

Stellenbosch University https://scholar.sun.ac.za

Contents

Declaration **ii**

Abstract **iii**

Uittreksel **v**

Acknowledgements **vii**

Contents **viii**

List of Figures **xii**

List of Tables **xv**

1 Introduction **1**

1.1 Problem Description 2

1.1.1 Satisfaction of Coverage Metrics 2

1.1.2 Adequate Evaluation 2

1.1.3 Assessing Student Submissions 3

1.2 Research Objectives 3

1.2.1 Development of Testing Framework 3

1.2.2 Implementation of Existing Testing Methods 4

1.2.3 Evaluation of Implemented Techniques 4

1.3 Structure of Thesis 4

Stellenbosch University https://scholar.sun.ac.za

2 Background **7**

2.1 Testing 7

2.1.1 Black-box Testing 7

2.1.2 White-box Testing 8

2.1.3 Test Inputs 9

2.2 Context-Free Grammars 10

2.3 LR Parsers 14

2.3.1 Handles 15

2.3.2 Computing Rightmost Derivations 16

2.3.3 LR Parsing Algorithm 20

2.4 Coverage Criteria 21

2.4.1 Positive Coverage Criteria 21

2.4.2 Negative Coverage Criteria 23

2.5 Related Work 24

2.5.1 Context Dependant Rule Coverage 24

2.5.2 LR(k) Parser Construction 25

3 Test Case Generation **27**

3.1 Purdom 27

3.1.1 Phase 1: Shortest String Algorithm 27

3.1.2 Phase 2: Shortest Derivation Algorithm 30

3.1.3 Phase 3: Sentence Generation 31

3.2 Zelenov and Zelenova 38

3.2.1 Preliminaries 38

3.2.2 PLL 40

3.2.3 WPLR 41

3.2.4 NLL 42

4 LR-Automata Search **45**

4.1 Breadth First Search 48

4.2 Breadth First Search with Termination 49

Stellenbosch University https://scholar.sun.ac.za

4.2.1 Search 49
4.2.2 Completion of Test Cases 50

5 Tool Implementation 53
5.1 Conversion From Input to BNF 54
5.1.1 Input to EBNF 54
5.1.2 EBFN to BNF 55
5.2 Sentence Construction 56
5.3 LR(1) Parse Table Generation 57
5.4 Output 60

6 Evaluation and Interpretation 61
6.1 Classroom Evaluation 61
6.1.1 Methodology 62
6.1.2 Case Study: Niklaus 63
6.1.3 Case Study: Simpl 72
6.1.4 Interpretation 77
6.2 Performance 78
6.2.1 Methodology 78
6.2.2 Results 80

7 Conclusion 87

Bibliography 89

A Simpl 91

B Niklaus 94
B.1 Textual Description 94
B.1.1 Structure 94
B.1.2 Declarations 94
B.1.3 Types 95
B.1.4 Statements 95

Stellenbosch University https://scholar.sun.ac.za

B.1.5 Arithmetic Expressions 95
B.1.6 Boolean Expressions 96
B.1.7 String Literals 96
B.1.8 Function Application Expressions 96
B.1.9 Comments 96
B.2 CFG 97

C Results: Nikluas **98**

D Results: Simpl **102**

Stellenbosch University https://scholar.sun.ac.za

List of Figures

2.1 An example of a CFG 11
2.2 Graph showing connections between nonterminals 13
2.3 Derivation tree for grammar 2.1 15
2.4 The LR(0) automaton computed for the example grammar 19

3.1 Derivation tree for rule 2, $E \rightarrow E + T$ 30
3.2 Visual example of line 15 in phase 2 of Purdom's algorithm 32
3.3 Construction of set of sentential forms for derivation chain for F to S 41

4.1 Parsing automaton for running example2.1 47
4.2 Infinite loop during the completion of a test case 51

5.1 Program structure 53
5.2 CFG to illustrate the structure of the EBNF grammar representation 55
5.3 EBNF representation of the *write* and *program* rules 55
5.4 Grammar after the all the rules have been expanded 56

6.1 Niklaus (N=63): Distribution of pass rates for the handwritten test cases 65
6.2 Niklaus (N=63): Distribution of the pass rates attained for a test set generated using Purdom's algorithm 66

Stellenbosch University https://scholar.sun.ac.za

6.3 Scatter plot showing the correlation between handwritten pass rates and those attained for a test set generated using Purdom's algorithm. The darker dots indicate that more students achieved the same pass rate for both test sets. 67

6.4 Niklaus (N=63): Distribution of the pass rates attained for a test set that satisfies the PLL coverage criteria. 67

6.5 Scatter plot showing the correlation between handwritten pass rates and those attained for a test set generated that satisfies PLL coverage. 68

6.6 Niklaus (N=63): Distribution of the pass rates attained for a test set that satisfies the WPLR coverage criteria. 68

6.7 Scatter plot showing the correlation between handwritten pass rates and those attained for a test set generated that satisfies WPLR coverage. 69

6.8 Distribution of the pass rates attained for the PLR test input set . 70

6.9 scatter plot showing the correlation between handwritten pass rates and those attained for the PLR test input set. 71

6.10 Distribution of the pass rates attained for a test set that satisfies the NLL coverage criteria. 71

6.11 Distribution of the pass rates attained for a test set that satisfies the NLL coverage criteria in the range $[98, 100]$. 72

6.12 Distribution of the pass rates attained for a test set that satisfies the NLR coverage criteria. 73

6.13 Distribution of the pass rates attained for a test set that satisfies the NLR coverage criteria in the range $[98, 100]$. 73

6.14 Bar graphs showing pass rates for each student over the handwritten test cases. 75

6.15 Pass rates for each submission evaluated with each set of generated positive test cases 76

6.16 Pass rates for each students evaluated with the negative generated test cases 76

Stellenbosch University https://scholar.sun.ac.za

6.17 Scatter plot showing the correlation between the running time of Purdom's algorithm and the grammar size 84
6.18 Scatter plot showing the correlation between the running time of the PLL and WPLR algorithms and the grammar size 84
6.19 Scatter plot showing the correlation between the running time of the NLL algorithm and the grammar size 85
6.20 Scatter plot showing the correlation between the running time of the PLR and NLR algorithm and size of the parsing automata. . . 85

Stellenbosch University https://scholar.sun.ac.za

List of Tables

2.1 Test Case Results 10
2.2 Reduction of the string id $*$ id to the start symbol 16
2.3 The shift-reduce method of parsing id $*$ id 17
2.4 LR(0) parsing of the string 'id $*$ id' 21

3.1 Values of RLEN, SLEN and SHORT after completion of phase 1 of Purdom's algorithm 29
3.2 DLEN and PREV after the completion of the second phase of Purdom's algorithm 32

4.1 Parse Table for running example 2.1 46

6.1 Number of test cases generated using each algorithm for each CFG 63
6.2 The grammars used for testing 80
6.3 Performance of the algorithm over the LL(*) grammars 81
6.4 Size of each test set constructed for the grammars. 82
6.5 Performance of PLR and NLR test case generation methods over the LR(1) grammars 83

Stellenbosch University https://scholar.sun.ac.za

List of Algorithms

2.1 Computation of LR(0) automaton 18
2.2 The LR parsing algorithm 20
3.1 Purdom's Algorithm: Shortest Terminal String 28
3.2 Purdom's Algorithm: Shortest Derivation 31
3.3 Purdom's Algorithm: Phase 3 Auxiliary Methods 36
3.4 Purdom's Algorithm: Sentence Generation 37
3.5 Construction of $first_n(\alpha)$ 39
3.6 Construction of a derivation chain 39
3.7 Constructing a sentential form from a derivation chain 41
4.1 Breadth First Search Over a Finite State Machine 49
4.2 Breadth First Search over a FSM with termination conditions . . 50
4.3 Completion of test cases after breadth first search 51
4.4 Breadth First Search Generating Negative Test States 52
5.1 Construction of a terminal string for every nonterminal 57
5.2 Algorithm to compute the LR(1) closure for an item set I 58
5.3 Computation of the LR(1) *goto* function 59
5.4 Computation of the LR(1) parsing automata 59

Stellenbosch University https://scholar.sun.ac.za

Chapter 1

Introduction

While numerous advancements have been made in the field of software testng over the past decades, in practice, it is still the most of expensive part of the development process when designing and implementing large software systems. Large software componanies spend an average of 35% of their software development budget on quality assurance and testing [3] and this is expected to to rise to 40% by 2018.

Automating much of the testing process greatly reduces the time it takes to test software. This includes automating the generation and preparation of the test inputs used and the process of evaluating the software with respect to each test input. In situations where the input to some software component may be considered as structured data, the use of context free grammars (CFGs) has proven to be extremely useful as a method of generating test input. CFGs may be used to describe structures for a number of unrelated software systems ranging from URLs and markup languages to file formats and programming languages.

When CFGs are employed in the automatic testing of software, it is important to ensure that the generated test inputs exercise all relevant parts of the system under test. As such, research has been conducted that defines different criteria by which we may measure how well a set of test inputs covers the available input domain for the system to be tested. These coverage criteria allow us to generate test inputs with a reasonable degree of certainty that the test inputs are representitive of the input domain and adequetly test the components of the software system.

In this dissertation we investigate the use of CFGs as a mechaniscm for the generation of test inputs for software systems. We present a framework which may be used to generate test inputs from a CFG based on a number of coverage criteria. We also develop and implement a new method for the generation of test inputs based on the finite state automata which may be constructed for the family of LR parsers. Each test input method generation technique is evaluated as a means to validate parsers generated from handwritten grammars. They are also evaluated in terms of performance to assess which properties of a context free grammar affect their running time.

Stellenbosch University https://scholar.sun.ac.za

1.1 Problem Description

1.1.1 Satisfaction of Coverage Metrics

When generating test cases for software systems it is useful to define coverage metrics that provide a measure of how well the set of generated test cases covers the input domain of the system under test. For a parser the input space corresponds to all possible strings. More than half of the coverage metrics explored in this dissertation are defined to take into account the type of the parser for which test cases are being generated. When working specifically with LR parsers and the coverage metrics defined for them (WPLR, PLR and NLR [23]), we see that the criteria defined in [23] are defined in terms of the mechanisms employed to facilitate the parsing of grammars, WPLR being defined in terms of the items in a grammar and PLR and NLR being defined in terms of the parsing automata, yet the sets of test inputs that must be constructed to satisfy these coverage metrics are defined in terms of the CFG, not taking into account the structures used in their definitions. With PLR and NLR coverage in particular the definition provided for the set of test inputs to satisfy this metric is only guaranteed to satisfy the coverage metric after a certain, unknown, number of iterations. As such, a method of generating sets of test input that attempts to satisfy these coverage criteria that is based on the mechanisms used to define them is needed.

1.1.2 Adequate Evaluation

The original evaluations provided for generation techniques explored in this dissertation are not adequate for the purpose of an empirical comparison to each other or the method of LR-automata search. The testing performed in those papers should be expanded upon to provide accurate empirical data of the performance of those algorithms in order to conduct a detailed comparison between them and LR-automata search.

Purdom's Algorithm

P. Purdom, in his seminal paper [22] details testing performed on a set of eight LR grammars, the smallest of the eight having 11 states and 13 transitions and the largest having 371 states and 1356 transitions. This set of test grammars is too small. Purdom, also, does not provide any performance measurements for the algorithm. He does, however, provide a measurement of coverage measurement in terms of the number of states and transitions in the automata generated by his parser generator.

There have been subsequent implementations of the algorithm [8, 14] and with them more testing was performed using the algorithms. In [4] 28 grammars of varying size are used to test the algorithm. There is, however, no mention of mention of any performance testing that was performed using the algorithm.

Zelenov and Zelenova

The set of algorithms presented by S. V. Zelenov and S. A. Zelenova [23], are tested on only three grammars. While the sizes of the grammars tested are considerable (with an average of 237 tokens per grammar), not all the algorithms

Stellenbosch University https://scholar.sun.ac.za

mentioned in the paper are implemented and tested on the grammars. Performance data, however, is collected for each tested algorithm on each grammar.

1.1.3 Assessing Student Submissions

Examiners and lecturers at the university level are faced with a significant amount of time consuming work when assessing the student submissions for a compiler course. Much of this includes the manual construction of positive and negative test inputs which are used to assess the quality of student submissions. This process can be improved through the automation of test cases generation and the collection of the relevant data to perform the assessment of the submissions. Firstly, because the bulk of the preparatory work required by the examiner may be decreased and, secondly, because the automatic generation of test case has advantages over constructing them manually, including:

1. The amount of time required to generate test inputs is less than that to write them by hand.

2. Many more test inputs can be automatically generated than can be feasibly hand written.

3. The generated test cases remove human error.

4. The coverage criteria that must be satisfied by the sets of test inputs represent the language generated by the grammars in different ways, for example, the PLL and NLL coverage criteria specified by Zelenov and Zelenova specifically test particular symbols in the grammar for each test case. Through the addition metadata to the test cases specifying what part of the grammar is covered by each test input in terms of the coverage criteria, it may be easier for students and examiners to find mistakes. Contributing to more productive student/examiner interactions.

1.2 Research Objectives

The primary objective of this dissertation is the implementation and evaluation of multiple methods to generate test inputs from a given CFG in the confines of a general framework that facilitates the generation of the test inputs and allows the methods to be evaluated against one another.

1.2.1 Development of Testing Framework

We aim to develop a grammar based test input generation framework with which we may generate multiple sets of test inputs for a given CFG. This allows for the initial implementation of various generation algorithms with the possibility to expand the set of algorithms at any point in the future. This framework must also allow for the addition and incorporation of the various auxiliary data structures required by the generation algorithms and various methods of outputting the generated test cases.

Stellenbosch University https://scholar.sun.ac.za

1.2.2 Implementation of Existing Testing Methods

We aim to satisfy four existing coverage criteria for CFGs and two coverage criteria that are defined in terms of LR-automata.

Existing Coverage Criteria

We aim to implement a number of algorithms described in two papers. First, P. Purdom's seminal paper in the field of grammar based test input generation [22], which constructs a set of test inputs that exercise every production rule in a grammar, is implemented as an example of simple production coverage. Second, the set of methods proposed by S. V. Zelenov and S. A. Zelenova [23] are implemented to explore more sophisticated coverage criteria. These methods construct sets of test inputs that satisfy the PLL, WPLR and NLL coverage criteria for CFGs. They are described in section 2.4.

Development and Implementation of a New Test Case Generation Algorithm Based on LR-Automata Traversal

We introduce a method for the generation of test input generation that is designed to use the finite state machine or parse table generated from a grammar to construct sets of positive or negative test inputs. First the finite state machine is searched using some generic search algorithm. This leaves us with a set of terminal strings that are incomplete as no string (except that string for which the breadth first search terminated at the end state) will form a sentence that can be derived from the start rule. These strings are then completed by appending to them the tokens that correspond to appropriate kernel items so that they may be reduced. Once this is done we may replace the nonterminals in each test case with an appropriate sequence of terminal tokens to produce a complete, valid test case.

1.2.3 Evaluation of Implemented Techniques

The implemented methods are evaluated against one another in two ways. Firstly they are used as a tool to assess handwritten parsers in a classroom setting. This substantially reduces the amount of work that must be done by instructors in creating test cases and marking student submissions by hand. Secondly they are evaluated against one another as a practical method for generating test inputs for a given software system. This includes measurements for the number of test cases generated, running time etc.

1.3 Structure of Thesis

This introductory chapter has provided context and an overview of the goals for this thesis. The rest of the thesis is organised in the following chapters.

Background We provide information relevant to the dissertation as follows: testing in Section 2.1, context free grammars in Section 2.2, coverage criteria in Section 2.4, LR parsers and the structures involved in their operation in Section 2.3 and related work in Section 2.5.

Stellenbosch University https://scholar.sun.ac.za

Test Case Generation We give an overview of the two existing methods of input test case generation:

1. An overview of Purdom's algorithms is given. Splitting the algorithm into 3 separate phases as is done in [16].
2. The methods proposed by S. V. Zelenov and S. A. Zelenova are described here.

LR-Automata Search We describe the method for the searching of LR-automata in detail.

Implemention We give an overview of the tool and its implementation.

Evaluation This chapter details specifically the methods of evaluation and the results obtained from them.

Conclusion We give a summary of the dissertation, its results and contribution is provided.

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Chapter 2

Background

This chapter provides a background of the neccesary foundational work for this thesis. A general discussion of testing, related concepts and definitions are given. Context free grammars, their associated algorithms, and their coverage criteria are defined. A foundational discussion of LR parsing and the operations performed by a parser is given. A summary of some important research done in all fields is given. For more information the reader is refered to [2, 25, 18].

2.1 Testing

Testing is the process of systematically exercising a software system in an attempt to verify that the system operates correctly within the confines of the set software requirements or to find test input that result in the incorrect operation of the system so that the flaw may be found and fixed. Here we provide definitions of the terms used throughout this dissertation to describe the quality of the tested software systems and give an overview of software testing.

When attempting to validate the stability and correctness of a software system we can consider the two perspectives of white-box and black-box testing.

2.1.1 Black-box Testing

We perform black-box testing by testing the system without consideration or knowledge of its implementation, making sure that the analysis of the sets of inputs given to the software adequately cover the *input space* of the of the software system. The software developed for this dissertation takes, as its input a context free grammar, and produces as output a set of sentences that are designed to satisfy specific coverage criteria. While the ultimate goal, to attain complete satisfaction that the software system performs as it is designed, is to test it with every possible value in the input space, this is not feasible as most software have an infinite input domain. This means it is impossible to be completely certain that there a no errors in a large software system. To avoid this and ensure that the system has been adequately tested, some techniques are used when choosing which test cases to use. More sophisticated methods for choosing test cases increase the likelihood that testing will reveal errors in

Stellenbosch University https://scholar.sun.ac.za

the software. Two methods that may be used when black-box testing software are equivalence partitioning and boundary-value analysis:

Equivalence Partitioning The process of partitioning the input domain for a software system into classes where inputs from each class, while not being identical, exhibit the same structural components is known as equivalence partitioning. These test cases, which have similar characteristics, will exercise the software system in a similar way. If we consider software that performs simple arithmetic calculations as an example, we can divide the input space into additions, subtraction, multiplication and division. These divisions are the equivalence partitions. This way instead of the software with as many random test cases as we can construct, we test with a small number of test cases from each equivalence partition and we can be confident that the software will perform correctly for all input from the same partition. In many instances equivalence partitions may be combined. The normal arithmetic operations may be combined to produce outputs with different results depending on the order in which the operations are applied. Testing may be performed with sets of test cases that fall into multiple equivalence partitions.

Boundary-Value Analysis When testing we are much more likely to find errors in the software if we consider not only the structure of the input we intend to use for testing, but also the expected output. This allows us to construct test cases that test the software at the extremes of the input domain. When considering the example of arithmetic software, we may use our knowledge of the input space to construct test inputs where we use the identity operators under the arithmetic operations Also test cases intended to produce adverse results, such as division by zero.

2.1.2 White-box Testing

White-box testing, on the other hand, is performed with the specifics of the implementation of the software system taken into consideration. With access to the internals of the system, we can construct test inputs that are intended to exercise or cover specific sections of the system. The ultimate goal of white-box testing to generate test inputs that exercise every possible path that may be taken in execution of the system under test. As with black-box testing this goal is not feasible as in practice, most large software systems present us with an infinite number of execution paths that may be exercised. To avoid attempting to satisfy this condition we may construct test inputs to satisfy coverage metrics with respect to the structure of the software.

Statement Coverage Test inputs are constructed in such a way that each statement in the system under test is executed at least once. This is a simple, and not very strong test criteria to satisfy. It does not take into account in what state the system may be in when a statement is executed.

Branch Coverage The set of constructed test inputs must cover each branch of every branching statement in the software system. This is a more sophisticated metric to satisfy. Satisfaction of branch coverage will satisfy statement coverage in all but 3 cases:

Stellenbosch University https://scholar.sun.ac.za

1. Programs that do not have any statements.
2. Programs where methods have multiple entry points.
3. Programs that have exception handling code that is not necessarily executed.

With these exceptions it is reasonable to define branch coverage as a set of test cases where each conditional statement in a software system has a true and false outcome, and that also satisfies statement coverage.

Condition Coverage Branch coverage only takes the ultimate outcome of a conditional statement into account. This is sufficient for simple conditional statements, but when we consider compound branching statements in a software system, it is not. A test input set that satisfies condition coverage, a stronger coverage criteria, is made up of tests that are constructed in such a way that every condition in a branching statement is true and false at least once and branch coverage is achieved. A stronger requirement is multiple-condition coverage. Here tests are constructed so that every combination of true/false values in a branching statement is exercised.

2.1.3 Test Inputs

Here we give an overview of the use of test inputs when testing a software system and describe how they may be used to specifically quantify the quality of parsers in the context of this dissertation.

To ensure the proper operation of a software system we must confirm that the software correctly processes the input that is provided to it. In the context of using a parser to determine whether an input sentence is in the language generated by the grammar it parses, we can expect only true or false values to be returned. These values indicate that an input provided to the parser is, or is not in the language the parser was designed to parse. General definitions for the input domain, test inputs and outputs are given:

Input Domain The set of all inputs that may be used with a software system. The input domain of parser is the set of all finite string from a given CFG.

Test Input A value from the input domain. For a parser this may be any single string in the input domain.

True Condition The absolute condition of the test input i.e., whether it is inherently positive or negative. The true condition of a parser test input indicates whether it is in the language generated by the grammar or not.

Predicted Condition The condition of test input that is predicted by the system.

Positive Test Input A test input in the input domain which we expect the system to accept i.e., the true condition is true.

Negative Test Input A test input in the input domain which we expect the system to reject i.e., the true condition is false.

True Positive (TP) The predicted condition of the test is true and the true condition of the test is true.

Stellenbosch University https://scholar.sun.ac.za

False Positive (FP) The predicted condition of the test input is true and the true condition of the test is false.

True Negative (TN) The predicted condition of the test input is false and the true condition of the test is false.

False Negative (FN) The predicted condition of the test input is false and the true condition of the test is true.

To quantify the quality of a parser we use the ratio of the number of passing test inputs and the total number of inputs provided to the system. The passing test inputs is the collection of true positive and true negative test inputs. The failing test inputs is the collection of false positive and false negative test inputs. This is shown in Table 2.1.

Predicted Condition	**True Condition**	
	Positive	Negative
Positive	True Positive	False Positive
Negative	False Negative	True Negative

Table 2.1: Test Case Results

The ratio of these values will be referred to as the pass rate (PR). It is also known as the accuracy. It is defined as:

$$PR = \frac{TP + TN}{TP + FP + TN + FN}$$

2.2 Context-Free Grammars

Context-free grammars are formal grammars that use recursive production rules to describe a pattern of strings. The productions are described using a hierarchical structure where productions may be expanded using any available rule regardless of the context in which that rule appears.

For many grammars parsers may be automatically constructed and during this construction it is possible for flaws in the grammar to be brought to light. Context free grammars, when used in combination with a parser constructor also allow developers and language designers to add new rules to the grammar very easily, reducing development time. The definitions and notation used to describe CFGs are mostly taken from [2].

Definition 2.1. *A context free grammar G is defined as a 4-tuple $G = (T_G, N_G, P_G, S_G)$ where:*

1. *T_G is a finite set. The elements of T_G are called the terminals of G.*
2. *N_G is a finite set, disjoint from T_G. The elements of N_G are called the nonterminals of G.*
3. *P_G is a finite relation from P_G to $(T_G \cup N_G)^*$. The elements of P are called the productions of G.*

Stellenbosch University https://scholar.sun.ac.za

4. S_G is a member of S_G. This is the start symbol of G.

The following notation will be used throughout this thesis. This notation is similar to the notation used in the popular book [2].

1. Terminal symbols will be, unless otherwise stated, denoted by a lower case letter, e.g. x, or a boldface string, e.g **id**. Any operator symbols (+, -, % ...), punctuation or digits, unless otherwise stated, will also denote a terminal symbol.

2. Nonterminals will be, unless otherwise stated, denoted by an uppercase letter e.g. X or a lowercase italic string e.g. *program*.

3. A string of grammar symbols, terminals or nonterminals, will be denoted by a lower case Greek letter e.g. α, β. These will be referred to as sentential forms.

4. A production rule will be denoted a uppercase letter and an arrow followed by a string of symbols, such as, $A \to \alpha$. The nonterminal, A, on the left-hand side of a production rule, $A \to \alpha$, is called the head of the rule and the sentential form, α, on the right-hand of the rule is called the body of the rule.

5. Unless it stated otherwise the head of the first rule in a grammar is the start symbol.

6. Multiple production rules with the same nonterminal symbol at the head may be written in the same line and separated by the '|' symbol e.g. $A \to \alpha_0 | \cdots | \alpha_n$.

A simple example of a context free grammar is given in Figure 2.1. Here $V_G = \{S,\ E,\ T,\ F\}$, $T_G = \{\mathbf{id},\ (,\ ),\ +,\ *\}$ and $S_G = S$. P_G is given by the rules of the grammar as shown in the figure. The production rules in the grammar are numbered 1–7. This grammar defines simple arithmetic expressions using addition and multiplication. It will also be used throughout the dissertation to illustrate all the implemented algorithms.

$$
\begin{array}{l}
(1)\ S \to E \\
(2)\ E \to E + T \\
(3)\ E \to T \\
(4)\ T \to T * F \\
(5)\ T \to F \\
(6)\ F \to \mathbf{id} \\
(7)\ F \to (\ E\ )
\end{array}
$$

Figure 2.1: An example of a CFG

Stellenbosch University https://scholar.sun.ac.za

Sentence Derivation

Deriving a string is the process of applying production rules, starting with the start symbol, until only terminal symbols remain. The application of a rule is denoted by $\alpha \implies \beta$. We may derive the symbol, **id** in the example grammar, with the derivations, $S \implies E \implies T \implies F \implies \mathbf{id}$. A sequence of zero or more rule applications is denoted by $\alpha \overset{*}{\implies} \beta$, and a sequence of one or rule applications is denoted by $\alpha \overset{+}{\implies} \beta$.

The set of all strings that that may be derived from the start symbol in a grammar is called the language generated by the grammar. This is denoted by $L(G) = \{\omega \in T_G^*|\ S \overset{*}{\implies} \omega\}$.

The application of rules may be performed in any order to derive a sentence, but we will use one of two methods of choosing which rule to apply in this dissertation, namely leftmost and rightmost derivation:

Leftmost derivation The leftmost nonterminal in a sentential form used in a derivation is always chosen for a rule application. Consider deriving the string $\mathbf{id} * \mathbf{id}$. It will be done by the following derivation:

$$\begin{aligned} S &\implies E \implies E * T \implies T * T \implies F * T \\ &\implies \mathbf{id} * T \implies \mathbf{id} * F \implies \mathbf{id} * \mathbf{id} \end{aligned}$$

A leftmost derivation is indication by $\alpha \underset{lm}{\implies} \beta$.

Rightmost derivation The rightmost nonterminal in a sentential form used in a derivation is always chosen for a rule application. Consider deriving the string $\mathbf{id} * \mathbf{id}$. It will be done by the following derivation:

$$\begin{aligned} S &\implies E \implies E * T \implies E * F \implies E * \mathbf{id} \\ &\implies T * \mathbf{id} \implies F * \mathbf{id} \implies \mathbf{id} * \mathbf{id} \end{aligned}$$

A rightmost derivation is indication by $\alpha \underset{rm}{\implies} \beta$.

Proper Grammars and Equivalence

A proper grammar is a CFG that does not exhibit any of the following traits.

Definition 2.2. *A context free grammar $G = (T_G, N_G, P_G, S_G)$ is a proper grammar if:*

1. *All nonterminal symbols can be reached from the start symbol i.e, for every $A \in N_G$ there exist the strings α, β such that we may be derive $S \overset{*}{\implies} \alpha A \beta$.*

2. *All nonterminals derive a string in T_G^* i.e., for every $A \in N_G$ there exists some $\omega \in T_G^*$ such that we have $A \overset{*}{\implies} \omega$. Nonterminals that do not derive a string in T_G^* are called unproductive symbols.*

3. *There are no empty productions. For every $A \in N_G$ there does not exist a rule $A \to \epsilon$.*

4. *There are no cycles. For every $A \in N_G$ the derivation $A \overset{+}{\implies} A$ does not exist.*

Stellenbosch University https://scholar.sun.ac.za

Multiple grammars may generate the same language. For any two grammars, G_1 and G_2, where $L(G_1) = L(G_2)$, we say that the grammars are weakly equivalent. Any grammar that does not produce ϵ may transformed into a weakly equivalent proper grammar. For grammars that do produce ϵ we may transform it into a weakly equivalent proper grammar where $S \implies \epsilon$ is the only rule that produces ϵ. We assume that all the grammars used throughout this dissertation exhibit all the traits of a proper grammar with the exception of not having any empty productions.

Successors and Derivation Chains

Nonterminals in a CFG are connected to each other through the derivations performed when building strings. The connections between the nonterminal symbols may be described by a directed graph. Consider the example grammar. Each symbol $A \in N_G$ represents a vertex in the graph and for every x there is an edge $(A,\ B)$, if there exists a rule $B \to \alpha A \beta$. This graph is shown in figure 2.2. Note the circular nature of the graph. With this we define successors

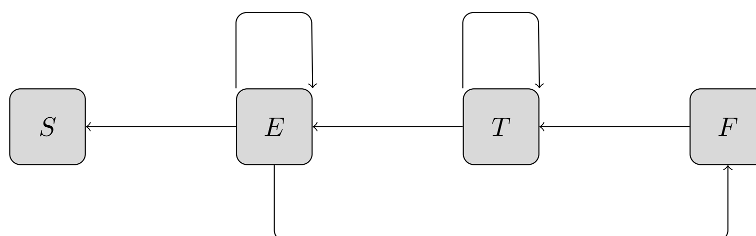


Figure 2.2: Graph showing connections between nonterminals

and derivation chains. This tells us which derivation to perform to derive one nonterminal from an other in as few derivation steps as possible.

Definition 2.3. *Let G be a context free grammar. If for a nonterminal, $A \in N_G$, there exists a rule $B \to \alpha A \beta$, then A is a successor of B. This is denoted by $A < B$.*

For every nonterminal, A, in our grammar we can construct a chain of successors, $A < A_0 < \cdots < A_k < S$, from it to the start rule. It is shown in [23] that for each nonterminal we can construct the shortest derivation chain from A to S, and that each such chain, if there are more than one, is unique. The method of using a derivation chain to construct a sentence is shown in chapter 3.2.2. This derivation chain is useful when using a nonterminal to construct sentential forms in the language. We can construct them by reverse derivation. Consider constructing a sentential form for the symbol F in the example grammar. This symbol produces the derivation chain $F < T < E < S$. We can construct sentential forms using the nonterminals in the chain from left to right as follows:

$$F \implies \{F,\ T * F\} \implies \{F,\ E + F,\ T * F,\ E + T * F\}$$

Stellenbosch University https://scholar.sun.ac.za

First and Follow Sets

The first set for a grammar symbol, x, *first*(x), is defined as the set of terminals symbols that can occur as the first symbols in any string of terminals derivable from x.

Definition 2.4. *Let G be a context free grammar. For any symbol , $x \in N_G \cup T_G$), we define $first(x) = \{t \in T_G | \ x \stackrel{*}{\Longrightarrow} t\beta\}$*

For a grammar symbol, x, *follow*(x), is the set of terminal symbols that can appear immediately after x in some sentential form.

Definition 2.5. *Let G be a context free grammar. For any grammar symbol, $x \in (N_G \cup T_G)$, we define $follow(x) = \{t \in T_G | \ S \stackrel{*}{\Longrightarrow} \alpha x t \beta\}$.*

To illustrate this the example grammar may be considered, where we see that $first(E) = \{\mathbf{id}, \ (\}$ and $follow(E) = \{+, \ )\}$.

Derivation Trees and Ambiguity

When deriving a string from a grammar it useful to construct a derivation tree. A derivation tree, also known as a parse tree, is an ordered tree that represents the steps taken during a derivation, allowing us to consider the derivation without having to consider the order in which the rules are applied to complete the derivation. The root of the derivation tree is the left hand side of the root rule used in the derivation. The leaves of the derivation tree are the terminal symbols that result from the derivation and all other nodes in the tree are non-terminal symbols used throughout the derivation. Nodes that are connected in the tree represent the rules that were used in the derivation. Using the example grammar Figure 2.3 shows the derivation tree for the string $\mathbf{id} * \mathbf{id} + \mathbf{id}$. The derivation of the string may be performed by applying the appropriate rules in many different combinations but, the derivation tree does not include this information.

A grammar that produces more than one derivation tree for the same derivation is said to be ambiguous. This happens when a grammar produces more than one leftmost or rightmost derivation for the same sentence. Consider the grammar $A \rightarrow aA \mid Aa \mid \epsilon$. This represents a string consisting of any number, including zero, of 'a' symbols. For the string 'aa' this grammar produces the two leftmost derivations:

$$A \Longrightarrow aA \Longrightarrow aa$$
$$A \Longrightarrow Aa \Longrightarrow aa$$

This makes the grammar ambiguous. To remove the ambiguity the grammar may be rewritten as $A \rightarrow aA \mid \epsilon$ or $A \rightarrow Aa \mid \epsilon$. In this dissertation we assume all grammars to be unambiguous.

2.3 LR Parsers

The primary purpose of any parser is the syntactic validation of a string with respect to some input grammar. The parser determines whether the given input

Stellenbosch University https://scholar.sun.ac.za

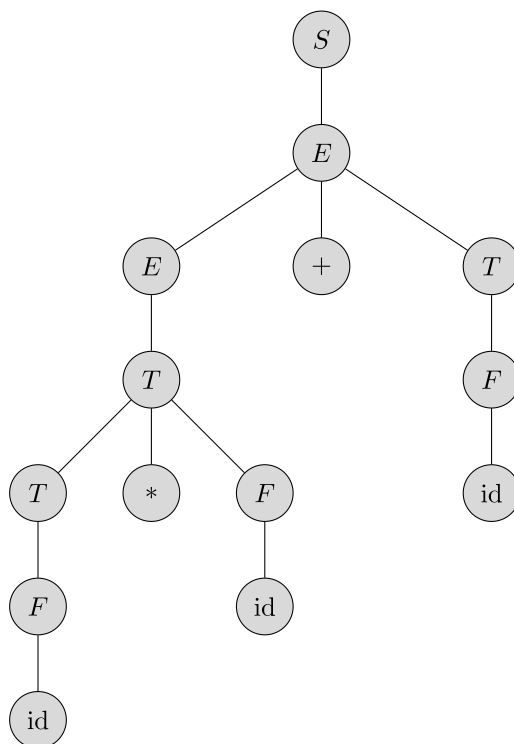


Figure 2.3: Derivation tree for grammar 2.1

string is in the language generated by the grammar which it was built to parse. LR parsers read the input from left to right and attempt to build a derivation tree for the string using rightmost derivations. The LR parser identifies the rules to apply to the derivation only once all constituents of the rule have been read, unlike the LL parser that attempts to identify the appropriate rule to apply before the constituent symbols of the rule have been read. It has been shown that this class of parsers are more powerful that LL parsers, with any LL grammar being LR(k) and LR(1) parsers being equivalent in recognition power to LR(k) parses for $k > 1$.

This section will provide a description of shift/reduce parsing. The algorithms to construct item sets and parsing tables will given in chapter 5.3. For more information the reader is referred to [2].

2.3.1 Handles

Because LR parsers validate a string by building the derivation from the bottom up (from the terminal symbols to the start symbol), the rightmost derivation for any string must be constructed in reverse. A rightmost sentential form is any sentential form, α, that may be derived by $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha$. To produce a rightmost derivation, as input is read by the parser and a sentential form is constructed, the parser applies grammar rules to the partial sentential form, replacing strings that fit the right hand side of a production with the left hand side. This is known as reduction. Reductions on the partial sentential form are performed as input

Stellenbosch University https://scholar.sun.ac.za

is read until the input has been reduced to the start symbol, at which the parse is complete. Table 2.2 shows the process of reduction on the string id $*$ id in the language of the example grammar.

Symbols Read	Sentential Form	Remaining Input
		id $*$ id
id	id	$*$ id
id	F	$*$ id
id	T	$*$ id
id $*$	$T*$	id
id $*$ id	$T*$ id	
id $*$ id	$T*F$	
id $*$ id	T	
id $*$ id	E	
id $*$ id	S	

Table 2.2: Reduction of the string id $*$ id to the start symbol

The method the parser uses to identify when to read a symbol, known as a shift, and when to apply a grammar rule, known as a reduction, is called handle finding. A handle may be described as a substring in a sentential form where the reduction of the string represents one step in the rightmost derivation of the sentential form.

Definition 2.6. *If* $S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A \beta \underset{rm}{\Longrightarrow} \alpha\omega\beta$*, then the rule* $A \rightarrow \omega$*, at the position after* α *is a handle of* $\alpha\omega\beta$*.*

Being able to identify handles and, more importantly, when to apply rules and when to shift more symbols is the primary method of operation of any LR parser.

2.3.2 Computing Rightmost Derivations

LR parsers employ the method of shift-reduce parsing, a form of stack-based parsing in which the stack holds the partial sentential form corresponding to the rightmost derivation of the input that has been read and an input buffer holds the remaining input to be consumed by the parser. This form of parsing has four basic operations:

1. **Shift**: The parser reads one symbol from the input buffer and pushes it on to the stack. It *shifts* a symbol from the input buffer to the stack.

2. **Reduce**: The parser identifies that a substring at the right end of the stack corresponds to a grammar rule. It pops off the substring and pushes the nonterminal at the left hand side of the grammar rule on to the stack. The substring corresponding to the right hand side of the rule is *reduced* to the nonterminal at the left hand side.

3. **Accept**: The parser stops the parsing process. This is done when the input string has been successfully reduced to the start symbol.

Stellenbosch University https://scholar.sun.ac.za

4. **Error**: The parser discovers a syntax error in the input and starts an appropriate error recovery routine or terminates the parse.

Table 2.3 shows how these operations are applied to the string 'id $*$ id' in the example grammar. The parser first marks the input buffer at the start and end with a symbol \$, which is not in the grammar. This symbol is used to mark the end of the input buffer so that the parser does not attempt to perform any shift operations when there is nothing to shift, also to identify an error if the stack has been reduced to the start symbol with more input to read in the input buffer. The LR parser makes shift-reduce decisions by means of an

Stack	Input	Action
\$	id $*$ id \$	shift id
\$ id	$*$ id \$	reduce by $F \to$ id
\$ F	$*$ id \$	reduce by $T \to F$
\$ T	$*$ id \$	shift $*$
\$ $T*$	id \$	shift id
\$ $T*$id	\$	reduce by $F \to$ id
\$ $T * F$	\$	reduce by $T \to T * F$
\$ T	\$	reduce by $E \to T$
\$ E	\$	reduce by $S \to E$
\$ S	\$	accept

Table 2.3: The shift-reduce method of parsing id $*$ id

automaton that keeps track of where the parser is in the current parse. The states of the automaton are made of sets of grammar rules marked at the place we expect to be in the current parse. These marked grammar rules are called items. An item is a grammar rule where the body is marked by some symbol not in the grammar and not \$, for convenience this symbol is denoted by a $\boldsymbol{\cdot}$. A single grammar rule produces items marked at every possible position in the rule. The rule $E \to E + T$ produces the items

$$E \to \cdot E + T$$
$$E \to E \cdot +T$$
$$E \to E + \cdot T$$
$$E \to E + T \cdot$$

The item $E \to E \cdot +T$ tells us that we have seen a string in the input that may be derived for E and we expect to see in the input buffer a string that can be derived from $+T$. The collection of item sets known the canonical LR(0) item set will be used to illustrate the construction of an LR(0) parsing automaton for our example grammar. To construct the LR(0) automaton for any arbitrary grammar, the grammar must first be augmented with a new rule $S' \to S$, where S is the start rule and S' is nonterminal not in V. This is to ensure that the parser knows when to announce the acceptance of a string by making sure that the start rule is not part of the grammar and is not used in the body of any other rule. In this way if the input string can been reduced to that symbol

Stellenbosch University https://scholar.sun.ac.za

we can be absolutely sure that the parse has been completed successfully. Two functions, the *closure* and *goto* functions, must also be defined.

Definition 2.7. *If I is a set of items in a grammar G, then closure(I) is the set of items that can be constructed from the following two rules:*

1. *Every item in I is in closure(I)*
2. *If $A \rightarrow \alpha \cdot B\beta$ is in I, and $B \rightarrow \gamma$ is a production, then all productions of the form $B \rightarrow \cdot\gamma$ are added to closure(I).*
3. *Step 2 is repeated until no more items are added to I.*

The closure collects, given a set of items, all other items that the parser may expect to see during the parsing process. If $A \rightarrow \alpha \cdot B\beta$ is in closure(I), then the parser expects to see, in the input buffer, a string derivable from $B\beta$. We add all the productions with B at the head to closure(I) to ensure that all viable prefixes of $B\beta$ that are derivable from B are accounted for.

Definition 2.8. *goto(I, X), where I is an item set and $X \in T_G \cup N_G$ is a grammar symbol, is defined to be the closure of the set of all items $A \rightarrow \alpha X \cdot \beta$ such that $A \rightarrow \alpha \cdot X\beta$ is in I.*

The *goto* function defines the transitions in the automaton from one state to the next. When we calculate $goto(I, X)$ we get the item set that must be transitioned to from I under the input X. Algorithm 2.1 shows how to combine the $closure(I)$ and $goto(I, X)$ functions to create the LR(0) automaton for a grammar.

Algorithm 2.1 Computation of LR(0) automaton

```
1:  procedure LR(0) ITEMS(G)
2:      C = closure({S' → ·S})
3:      while new items are added to C do
4:          for each item set I ∈ C do
5:              for each grammar symbol X ∈ T_G ∪ N_G do
6:                  if goto(I, X) ≠ ∅ and goto(I, X) ∉ C then
7:                      C = C ∪ goto(I, X)
8:                  end if
9:              end for
10:         end for
11:     end while
12: end procedure
```

The algorithm begins by calculating the closure for the item set containing only the start symbol. This is the first state for the automaton. Then, until no new states can be added, the transitions and their corresponding item sets are computed for every state in the automaton. Using the example grammar figure 2.4 shows the result of the LR(0) item set construction algorithm. In each state we see two types of items:

Stellenbosch University https://scholar.sun.ac.za

1. The kernel items, shown at the top of each state, include the item $S' \to \cdot S$ and all other items that are of the form $A \to \alpha \cdot \beta$, where $|\alpha| > 0$. These are the items where the dot is not at the left of the rule

2. The non-kernel, shown at the bottom of each state in grey, are all items, excluding the start item, that are of the form $A \to \cdot\alpha$

Every state in an LR item set can be uniquely identified by the kernel items in the item set.

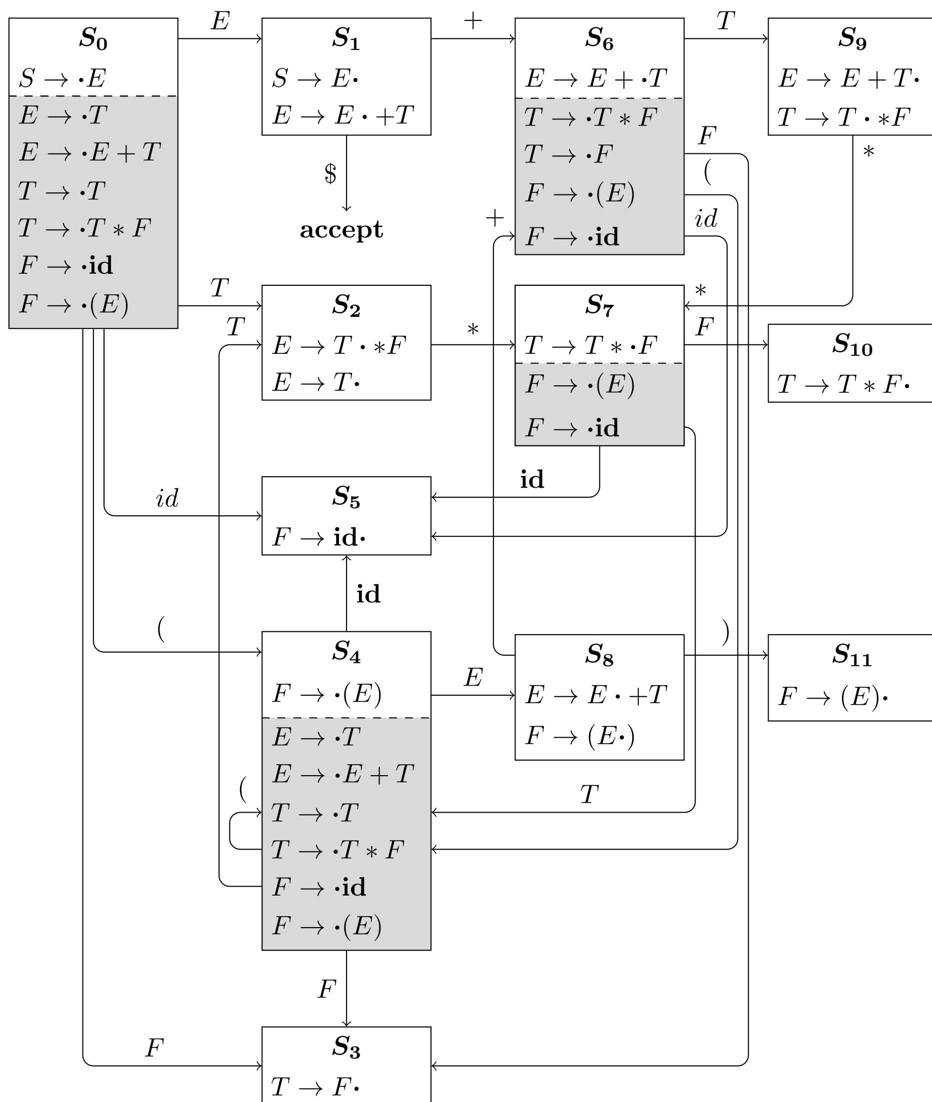


Figure 2.4: The LR(0) automaton computed for the example grammar

Stellenbosch University https://scholar.sun.ac.za

2.3.3 LR Parsing Algorithm

An LR parser uses the parsing automaton, a stack and an input buffer. Unlike the method of shift-reduce parsing discussed earlier, the LR parser does not shift grammar symbols on to the stack, but rather the states of the LR automaton. The algorithm for all LR parsers is the same, only the parsing automaton changes from one parser to the next. Algorithm 2.2 shows the algorithm that is used by an LR parser to validate input strings.

During the parse of a string, as the parser reads symbols and applies the shift and reduce operations, the state of the parser changes. For the shift-reduce parser we may define this state using the partial rightmost derivation and the remaining input with the pair:

$$(A_0A_1 \dots A_m, \alpha_0\alpha_1 \dots \alpha_n)$$

This pair uniquely defines each state that the parser may be in. For the LR parser, where the states are pushed onto the stack, we define it as

$$(s_0s_1 \dots s_m, \alpha_0\alpha_1 \dots \alpha_n\$)$$

where the first component represents the state stack of the parser and the second component the remaining input to be read. This is called the parser configuration. The following lines in algorithm 2.2 are noteworthy:

Algorithm 2.2 The LR parsing algorithm

1: **procedure** LR PARSING ALGORITHM(input string: α)
2: x = first symbol of α
3: **while True do**
4: s_0 = top of the stack
5: **if** $goto(s_0, x) \neq \emptyset$ **then**
6: push $goto(s_0, x)$ onto the stack
7: x = next input symbol
8: **else if** $\{S' \to S\cdot\} \in s_0$ **and** $x = \$$ **then**
9: **break**
10: **else if** s_0 has a rule $A \to \beta\cdot$ **then**
11: pop $|\beta|$ symbols of stack
12: t = top of stack
13: push $goto(t, A)$ onto the stack
14: **else**
15: report an error
16: **end if**
17: **end while**
18: **end procedure**

Lines 5 – 7 This if-statement represents a shift operation on a terminal symbol. If $goto(s_0, x) \neq \emptyset$ then the set computed by it must correspond to a state in the automaton and this is pushed on to the state stack.

Line 8 – 9 The parser checks the conditions for the acceptance of the input string. Firstly we check if $S' \to S\cdot$ is in s_0, which means we can reduce

Stellenbosch University https://scholar.sun.ac.za

to the start symbol, and secondly we must check that all input has been read by making sure the next input symbol is \$. If both conditions are true the parsing process may be terminated.

Lines 10 – 13 This if statement represents a reduction operation. If s_0 contains an item of the form $A \rightarrow \beta\cdot$ then we know we have read an entire rule and can now reduce. After the reduction we have popped off the stack all symbols that make up the rule and must take the transition that corresponds to the head of the rule.

Lines 14 – 15 If the parser cannot shift an input symbol, reduce the current states on the stack, or accept the input, there must be an error and the appropriate error reporting or recovery routing must be called.

Table 2.4 shows how an LR parser, using algorithm 2.2 in combination with the automaton shown in Figure 2.4 to parse the string 'id $*$ id'.

	Stack	Symbols	Input	Action
(1)	0	\$	id $*$ id \$	shift id
(2)	0 5	\$ id	$*$ id \$	reduce by $F \rightarrow$ id
(3)	0 3	\$ F	$*$ id \$	reduce by $T \rightarrow F$
(4)	0 2	\$ T	$*$ id \$	shift $*$
(5)	0 2 7	\$ $T*$	id \$	shift id
(6)	0 2 7 5	\$ $T*$id	\$	reduce by $F \rightarrow$ id
(7)	0 2 7 10	\$ $T * F$	\$	reduce by $T \rightarrow T * F$
(8)	0 2	\$ T	\$	reduce by $E \rightarrow T$
(9)	0 1	\$ E	\$	reduce by $S \rightarrow E$
(10)	0 1	\$ S \$		accept

Table 2.4: LR(0) parsing of the string 'id $*$ id'

2.4 Coverage Criteria

When generating sentences from context free grammars for the purpose of testing a software system we must ensure that the generated test inputs are representative of the input domain. While it is not feasible to generate every possible input in the domain for the system, as there may be an infinite number of unique inputs, we can, however, ensure that the generated test cases satisfy specific coverage criteria. In his way we can be reasonably certain that the set of test inputs is representative of the input domain for the software under test. The definitions for the LL and LR coverage criteria are taken from [23].

2.4.1 Positive Coverage Criteria

Here definitions and descriptions of the positive coverage criteria are provided and an example of a set of test inputs that satisfies the coverage criterion is given. For each of the following definitions let $G = (T_G, N_G, P_G, S_G)$ be a context free grammar and T be a set of sentences.

Stellenbosch University https://scholar.sun.ac.za

Terminal Coverage

Definition 2.9. *$T \subset L(G)$ satisfies terminal coverage if for every terminal symbol, $x \in T_G$, there is a sentence $t \in T$ such that $x \in t$.*

Terminal coverage is satisfied when each terminal symbol in the grammar is used as part of a sentence in T. We have for every terminal symbol some sentence, $t \in T$ that can be derived $S \stackrel{*}{\Longrightarrow} \alpha x \beta \stackrel{*}{\Longrightarrow} t$. In terms of the example grammar, where $T_G = \{\mathbf{id}, +, *, (,)\}$, the set, $T = \{(\mathbf{id} * \mathbf{id} + \mathbf{id})\}$, already satisfies the terminal coverage criterion.

Production Coverage

Definition 2.10. *$T \in L(G)$ satisfies production coverage if for every production rule $A \to \alpha \in P_G$ there is a sentence $t \in T$ such that $S \stackrel{*}{\Longrightarrow} \gamma A \beta \stackrel{*}{\Longrightarrow} \gamma \alpha \beta \stackrel{*}{\Longrightarrow} t$.*

Production coverage is satisfied when each rule of the grammar is exercised by a test input. In terms of the example grammar the set described above that satisfies terminal coverage also satisfies production coverage. We show how the sentence in T may be derived and number each production covered by the derivation steps:

$$\begin{aligned}
S &\stackrel{1}{\Longrightarrow} E \stackrel{3}{\Longrightarrow} T \stackrel{5}{\Longrightarrow} F \stackrel{7}{\Longrightarrow} (E) \\
&\stackrel{2}{\Longrightarrow} (E+T) \stackrel{4}{\Longrightarrow} (T*F+T) \stackrel{5}{\Longrightarrow} F*F+T \\
&\stackrel{6}{\Longrightarrow} (\mathbf{id}*F+T) \stackrel{6}{\Longrightarrow} (\mathbf{id}*\mathbf{id}+T) \stackrel{5}{\Longrightarrow} (\mathbf{id}*\mathbf{id}+F) \\
&\stackrel{6}{\Longrightarrow} (\mathbf{id}*\mathbf{id}+\mathbf{id})
\end{aligned}$$

We see that all the production rules 1–7 must be used to derive this sentence so production coverage is satisfied.

PLL Coverage

Definition 2.11. *Let i be the pair (A, x), where $A \in N_G$ and $x \in T_G$. The pair i is covered if there exists a $t \in T$ such that $S \stackrel{*}{\Longrightarrow} \alpha A \beta \stackrel{*}{\Longrightarrow} \alpha x \gamma \beta \stackrel{*}{\Longrightarrow} t$. T satisfies PLL (Positive LL) coverage if every pair in the set $\bigcup_{A \in N_G} \{(A, x),\ x \in$ first$(A)\}$ is covered.*

Positive LL coverage specifies that the set of sentences ensure that all nonterminals in the grammar are derived to sentential forms that start with each terminal symbol in their *first* set. In terms of the example grammar we note that the *first* set of every nontermial is $\{\mathbf{id}, (\}$. The set $\{\mathbf{id},\ (\mathbf{id})\}$ will satisfy PLL coverage for the example grammar.

WPLR Coverage

Definition 2.12. *Let i be the pair (π, x), where $x \in T_G$ and $\pi = A \to \alpha \cdot B \beta$ is an item in G. The pair i, is covered if there is a $t \in T$ such that $S \stackrel{*}{\Longrightarrow}$*

Stellenbosch University https://scholar.sun.ac.za

$\gamma A\delta \implies \gamma\alpha B\beta\delta \overset{*}{\implies} \gamma\alpha x\mu\beta\delta \overset{*}{\implies} t$ *The set T satisfies WPLR (Weak Positive LR) coverage if every pair of the form ($\pi = A \to \alpha \cdot x\beta, \sigma$), where $\sigma \in first(\pi)$) is covered.*

The WPLR coverage criteria specifies that the set of sentences ensures that all items in the grammar is derived are derived to sentential forms that start with each terminal symbol in the *first* set of the symbol after the dot. In terms of the example grammar the set $\{(\mathbf{id}), ((\mathbf{id}))\}$ covers the items $F \to (\cdot E)$, $E \to \cdot T$ and $T \to \cdot F$.

PLR Coverage

Definition 2.13. *Let i be the pair (s, x), where s is a state in the LR-automata, and $x \in T_G$. The pair i is covered if there is a $t \in T$ such that $S \overset{*}{\implies} \alpha x\beta \overset{*}{\implies} t$, where, after parsing α the parser has at the top its the stack the state s, and x is the next symbol to be read in the input string. PLR (Positive LR) coverage is satisfied when all pairs (s, x) are covered by the set of sentences T.*

PLR coverage specifies that every shift transition in the LR-automata must be exercised by the sentences in the set of test inputs. In terms of the example grammar, figure 2.4 shows an LR-automata that can be constructed for the example grammar. The sentence **id** covers the pair $(0, \mathbf{id}$, and the sentence $(\mathbf{id})$ covers the pairs $(0, ()$, $(4, \mathbf{id})$ and $(8,))$.

2.4.2 Negative Coverage Criteria

For the following to definitions defining the coverage of a set of negative test cases over a LL or LR parser, let $x \in T_G \cup N_G$ be some grammar symbol in G. The token, t, is called a feasible pretoken for x if the sentential form $\alpha t x\beta$ can be derived from the start symbol. t can also be called a feasible pretoken for the state s in the LR parsing the automaton if the sentential form $\alpha t\beta$ can be derived from the start rule, such that after reading the sequence αt the parser has the state s at the top of its state stack.

Let R_x be the union of the compliments of the follow sets for all pretokens of x.

$$R_x = \bigcup_t (T_G \setminus follow(t)) \qquad \text{where } t \text{ is a pretoken of } x$$

In other words R_x is the set of symbols that can not come before the symbol x in the language $L(G)$.

As an example consider the example grammar. The only feasible pretoken for the symbol T is $+$. To construct R_T we see that $follow(+) = \{(, \ \mathbf{id}\}$. We take the compliment of this with respect to T_G to get $R_t = \{*, \ )\}$

NLL Coverage

Definition 2.14. *Let i be the pair (A, t'), where t' is in the set R_A. The pair i is covered if the parser encounters the following situation when reading $\alpha t' A\beta$, A is the symbol on top of the stack and the incorrect symbol t' is the next token to be read. A test set T satisfies NLL coverage when all pairs (A, t') in G are covered.*

Stellenbosch University https://scholar.sun.ac.za

NLR Coverage

Definition 2.15. *Let i be the pair (s, t'), where t' is in the set R_s and s is a state in an LR parsing automata. The pair i is covered if the parser encounters the following situation when reading $\alpha t' \beta$. After the string α is read the parser must have the state s at the top of the state stack and the incorrect symbol, t', must be the next token to be read. A test set T satisfies NLR coverage when all pairs, (s, t'), in the grammar G are covered.*

2.5 Related Work

This section will discuss and summarise previous work that is directly related to the work developed and implemented in the dissertation. Section 2.5.1 provides a coverage metric for test case generation that exposes flaws in, and extends, production coverage. In section 2.5.2 a short summary is given of research in the field of LR parser construction.

2.5.1 Context Dependant Rule Coverage

The framework developed for this dissertation provides an implementation of Purdom's algorithm, a test generation method based on production coverage. Lämmel, in [13] shows that methods of test case generation based on this simple coverage criterion are inadequate at illuminating even the most simple errors in a parser. To remedy this a new coverage criterion, context dependant rule coverage (CDRC), is introduced to take into account the context of where the rule to be covered occurs. This is defined as follows:

Definition 2.16. *Let $G = (T_G, N_G, P_G, S_G)$ be a context free grammar. If $p = m \rightarrow \alpha n \beta \in P_G$, with $n \in N_G$, then the rule, $m \rightarrow \alpha n \beta$ is called a direct occurrence of n in G. $O_c(G, n)$ denotes the set of all direct occurrences of n in G.*

Once we have calculated the set of all occurrences for a nonterminal we can define CDRC.

Definition 2.17. *Let $G = (T_G, N_G, P_G, S_G)$ be a context free grammar and $T \subseteq L(G)$. $t \in T$ is said to cover the rule $p = n \rightarrow z$ for the occurrence $m \rightarrow \alpha n \beta$ in G if there is a derivation $S \overset{*}{\Longrightarrow} \gamma m \theta \overset{q}{\Longrightarrow} \gamma \alpha n \beta \theta \overset{p}{\Longrightarrow} \gamma \alpha z \beta \theta \overset{*}{\Longrightarrow} t$, where $q = m \rightarrow \alpha n \beta \in P$. T is said to cover $p \in P$ for all occurrences if there is a $t \in T$ that covers every occurrence $o \in O_c(G, n)$. T is said to satisfy context dependant rule coverage if T covers every $p \in P$ for all occurrences of p.*

The improvement of CDRC over naive production coverage is illustrated by Lämmel with the following example. Consider the following two grammars:

G_1	G_2
$s \rightarrow A\ B$	$s \rightarrow A\ B$
$A \rightarrow C\ a$	$A \rightarrow a$
$B \rightarrow b\ C$	$B \rightarrow b\ C$
$C \rightarrow \epsilon$	$C \rightarrow \epsilon$
$C \rightarrow c\ C$	$C \rightarrow c\ C$

Stellenbosch University https://scholar.sun.ac.za

We see that the string '*abc*' achieves rule coverage for both grammars, but does not show any discrepancy between them. The set of strings {*'bc', 'cabc'*}, which achieves CDRC for both grammars, shows that G_2 and G_1 do not generate the same language.

As a generalisation of production coverage the CDRC criteria is most closely related to the PLL coverage criteria. The concept of an occurrence is similar to that of a successor, which is used in the satisfaction of PLL. They differ in the fact that, as will be explained in Chapter 3, the successor is used to generate a sentential whereas the occurrence of a nonterminal is integral of the definition of CDRC itself.

The concept of context dependant rule coverage is sensitive to the manner in which a grammar is defined. In particular to grammars where there are chain rules, nonterminals which are defined by a single rule of the $n \to n'$. Consider the grammars:

G_3	G_4
$s \to A\ B$	$s \to A\ B$
$A \to C'\ a$	$A \to C\ a$
$B \to b\ C'$	$B \to b\ C'$
$C \to \epsilon$	$C \to \epsilon$
$C \to c\ C$	$C \to c\ C$
$C' \to C$	$C' \to C$

$L(G_4) = L(G_3)$, but the set $\{abcc\}$ can only satisfy CDRC for G_3, it is not sufficient for G_4 as the derivation of the string in G_4

$$s \implies AB \implies CaB \implies aB \implies abC' \implies abC \implies abcC \implies abcc$$

does not use the rule $C \to Ca$ where it occurs in $A \to Ca$.

Lämmel proposes a generalisation to CDRC, CDRC*, that remedies this by including in the definition the coverage of all direct and indirect occurrences of a symbol. An indirect occurence is defined as a sequence of direct occurrences. This generalisation solves the problem with chain rules and removes the dependency on the structure of the grammar.

2.5.2 LR(k) Parser Construction

The construction of LR(k) parsers, described by Knuth in [11], has been shown to be an inefficient process. The number of states needed by an LR parser are in the worst case exponential with respect to the size of the grammar. Korenjak [12] describes a method by which the speed of construction for an LR may be increased. This is done by partitioning the original grammar into a number of smaller grammars. If the original grammar is LR these smaller grammars must be LR as well and we can construct parsing tables for each. This construction uses less time and space than the construction for the original grammar. And the separate tables may be used together to construct a LR parser for a large grammar for which the construction of such a parser would not have been feasible.

Pager [19], presents conditions for the merging of states in the LR(k) finite

Stellenbosch University https://scholar.sun.ac.za

state machine. These conditions, known as the weak and strong compatibility conditions, allow us to shrink the of number of states the LR parsing automaton to somewhere in between the size of an LR(k) and SLR(k)/LALR(k) parser. The parser may be generated only using the weak compatibility condition, but when also applying the strong compatibility condition, we are guaranteed to generate a LALR(k) parser, if the target grammar is LALR(k), otherwise the algorithm will generate a LR(k) parser with less states than if it were produced by the canonical item set construction. This method is employed in *hyacc* [5], an LR(1) parser generator.

While the methods of optimising LR(k) parsers described above greatly decreased the time and space complexity of constructing a parser, they were still not enough to mitigate the inherently large size of the LR(k) parser. As a result the LALR(k) parser [6], with the optimisation in the computation of the look-ahead sets [7], has the predominantly used method of LR parsing in practice. There are multiple ways to construct an LALR parser, the most popular of them being the channel algorithm, described in detail in [9, 2], employed by the *yacc/bison* parser generator [10].

Stellenbosch University https://scholar.sun.ac.za

Chapter 3

Test Case Generation

This chapter will provide detailed descriptions of Purdom's algorithm and the methods of Zelenov and Zelenova to satisfy PLL, WPLR and NLL coverage.

3.1 Purdom

Purdom's algorithm describes a method of generating strings from a context free grammar quickly and in such a way that each rule in the grammar is used at least once. The algorithm was originally described in a very imperative fashion, with no high-level description or pseudocode to aid the reader. It was reformulated from two perspectives by Power and Mallow [16, 17] and these perspectives were further expanded upon by Paracha [20] and Butrus [4]. This description of Purdom's algorithm will draw from these sources. We will indicate and provide explanations for any deviations from them where necessary.

The algorithm itself may be split into three phases, with the first two phases being used as preparatory phases to generate crucial data that will be used by the third phase in the construction of the test sentences. To illustrate the operation of the algorithm we will use the example grammar shown in figure 2.1.

3.1.1 Phase 1: Shortest String Algorithm

This first phase of the algorithm computes the length of the shortest string derivable from a specific symbol and which production rule in the grammar is required to produce this derivation. Three data structures are needed for this phase:

SLEN This is a map that will store the length of the shortest string for each symbol in the grammar.

RLEN This is a map that will store the length of the shortest string that can be derived for each production rule in the grammar.

SHORT This is a map that will store, for each nonterminal, the number of the production rule that must be applied to derive the string with the shortest length.

Stellenbosch University https://scholar.sun.ac.za

SHORT is the primary deliverable of this phase and will be used again in phase 3 during the generation of the individual strings. Before the phase may begin SHORT, RLEN and SLEN must be initialised. For each terminal symbol $t \in T_G$, SLEN[t] is initialised to 1. For each nonterminal symbol $n \in N_G$, SLEN[n] is initialised to some maximum integer value, which we denote as infinity (∞), and SHORT[n] is initialised to -1. This initialisation, along with the rest of the algorithm is shown in algorithm 3.1.

Algorithm 3.1 Purdom's Algorithm: Shortest Terminal String

```
 1: procedure INIT PHASE 1(SLEN, RLEN, SHORT)
 2:     for t ∈ T_G : do SLEN[t] ← 1
 3:     for n ∈ N_G : do SLEN[n] ← ∞, SHORT[n] ← −1
 4:     for p ∈ P_G : do RLEN[p] ← ∞
 5: end procedure
 6: procedure SHORTEST STRING(SLEN, RLEN, SHORT)
 7:     change = True
 8:     while change is True do
 9:         change = False
10:         for p ∈ P_G do
11:             sum = 1, tooBig = False
12:             for each element e ∈ RHS[p] do
13:                 if SLEN[e] == ∞ then
14:                     tooBig = True, break
15:                 end if
16:                 sum += SLEN[e]
17:             end for
18:             if tooBig == False and sum < RLEN[p] then
19:                 RLEN[p] = sum
20:                 if sum < SLEN[LHS[p]] then
21:                     SHORT[LHS[p]] = p
22:                     SLEN[LHS[p]] = sum
23:                     change = True
24:                 end if
25:             end if
26:         end for
27:     end while
28: end procedure
```

There are a few things to note when inspecting algorithm in the following lines in algorithm 3.1:

Line 11 The variable sum is initialised to 1 and not 0. This is to prevent an infinite loop when generating strings in phase 3. It is necessary to define the shortest length of a string in a such a way that it increases when going around a series of recursive productions. To achieve this the length of the shortest string, which is the value used for SLEN, is defined in Purdom's original paper as the length of the string plus the number of steps needed in the derivation of that string.

Stellenbosch University https://scholar.sun.ac.za

Lines 12 – 17 This loop, over every symbol on the right hand side of the production p, is what calculates the length of the string produced by the right hand side of p. It uses the known values of the string lengths already calculated for SLEN. The if statement on line 13 is used to check that a value for e has already been calculated as all unknown values for SLEN are set to infinity.

Lines 14 and 18 On line 14 the variable tooBig is set to true if the symbol a string length for the symbol e has not been calculated. This indicates that the length of the string produce by the current production cannot be calculated. It is also used on line 18 to guard against the updating of RLEN, SLEN and SHORT with an incorrect value.

Lines 19 – 24 Once it has been confirmed that value calculated for the length of the production is valid this section of the algorithm updates the values for RLEN, SLEN and SHORT. If the length calculated is shorter than the already known length of the production RLEN is updated and if the length calculated for the production is smaller than the known length for the nonterminal on the left hand side of the production, SHORT and SLEN are updated.

It is useful to note that RLEN and SLEN both map their entries to the same value, namely the shortest string that can be derived from the entry. This means that for all the productions in RLEN produced by the same nonterminal, there will be a single entry in SLEN, mapping this nonterminal to the value of the shortest string derivable from it. This is shown in table 3.1. We see that the RLEN[1] and RLEN[2] corresponds to the rules in the grammar that have E as the left hand side. The shortest of these, RLEN[2], which is equal to 9, is used as the entry for SLEN[E], with SHORT[E] as the matching entry telling us what the rule number is to get this length. This is the case with all nonterminals in the grammar and there corresponding entries in RLEN, SLEN and SHORT.

SLEN	
S	5
E	4
T	3
F	2

RLEN	
1	5
2	9
3	4
4	7
5	3
6	2
7	7

SHORT	
S	0
E	2
T	4
F	5

Table 3.1: Values of RLEN, SLEN and SHORT after completion of phase 1 of Purdom's algorithm

With the adjusted definition of the shortest terminal string as the length of the string plus the number of steps required to derive it used in RLEN and SLEN, Power and Mallow note that this definition corresponds to the number of nodes we would have in a derivation tree for the string if we were to construct one. This is illustrated in figure 3.1 where we have 9 nodes in the derivation when the leftmost E symbol is taken as the root of the tree.

Stellenbosch University https://scholar.sun.ac.za

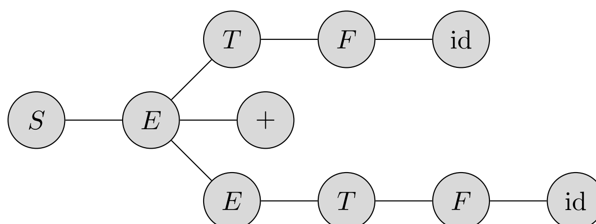


Figure 3.1: Derivation tree for rule 2, $E \to E\ +\ T$

3.1.2 Phase 2: Shortest Derivation Algorithm

The second phase of Purdom's algorithm computes two values for each nonterminal in the grammar. Firstly, it computes the length of the shortest derivation that uses a specific nonterminal, and it computes which production to use to introduce that nonterminal into the shortest derivation. This phase uses two data structures in combination with RLEN and SLEN calculated in the previous phase.

DLEN A list which, will upon the completion of this phase, contain for each nonterminal in the grammar, the length of the shortest derivation from the start symbol that uses that nonterminal.

PREV A list which, will upon the completion of this phase, contain for each nonterminal in the grammar, the number of the production rule to use when introducing that nonterminal into the shortest derivation.

DLEN and PREV will both be used in combination with SHORT, the primary deliverable of the previous phase, in phase 3 when generating sentences from the grammar. As with phase 1, the lists to be used in this phase must be initialised. For each nonterminal $n \in T_G$, PREV[n] is initialised to -1. For each nonterminal symbol $n \in T_G \setminus S$, DLEN[n] is initialised to ∞. This initialisation, along with the remainder of the algorithm is shown in algorithm 3.2.

The following sections of algorithm 3.2 are noteworthy:

Lines 2 – 6 This section of the algorithm contains the initialisation of DLEN and PREV. In [16] the corresponding section of the algorithm that provide does not contain the reassignment of DLEN[S] to SLEN[S] and no mention of it is made in the paper. If this step is not taken it results in an infinite loop as the check on line 14 will result in lines 15 – 22 being not being executed. We believe that this was a simple typographical error as it is corrected in their later top-down reformulation of Purdom's algorithm [17]. This assignment must be done as not only is the start symbol the root of all derivations, it is not introduced by any other symbol.

Lines 13 – 15 These conditional statements guard against the algorithm executing when there are no values for the production.

Line 16 This line calculates the length of the shortest derivation that contains the production rule p. It does so by taking the length of the shortest derivation that contains the left hand side of p (DLEN[LHS[p]]) and removing, from it, from the length shortest string that can be derived from

Stellenbosch University https://scholar.sun.ac.za

Algorithm 3.2 Purdom's Algorithm: Shortest Derivation

```
1: procedure INIT PHASE 2(SLEN, RLEN, DLEN, PREV)
2:     for n ∈ T_G do
3:         DLEN[n] = ∞
4:         PREV[n] = -1
5:     end for
6:     DLEN[S] = SLEN[S]
7: end procedure
8: procedure SHORTEST DERIVATION(SLEN, RLEN, DLEN, PREV)
9:     change = True
10:    while change = True do
11:        change = False
12:        for each production p ∈ P_G do
13:            if RLEN[p] == ∞ continue
14:            if DLEN[LHS[p]] == ∞ continue
15:            if SLEN[LHS[p]] == ∞ continue
16:            sum = DLEN[LHS[p]] + RLEN[p] - SLEN[LHS[p]]
17:            for each nonterminal n ∈ RHS[p] do
18:                if sum < DLEN[n] then
19:                    change = True
20:                    DLEN[n] = sum
21:                    PREV[n] = p
22:                end if
23:            end for
24:        end for
25:    end while
26: end procedure
```

the left hand side nonterminal from the derivation tree (-SLEN[LHS[p]]) and replacing it with the shortest string that can be derived from the current production (+RLEN[p]). Figure 3.2 shows an intuitive example of this using the derivation trees themselves.

Lines 17 – 22 Here the value of the new shortest derivation containing production p is compared to the existing values and DLEN and PREV are updated accordingly if the new value is less than the existing one.

Table 3.2 shows the values in DLEN and PREV after the execution of the second phase of the algorithm. Note that the length of the shortest derivation for every nonterminal in DLEN is 5. This is because the shortest derivation from the start symbol, $S \implies E \implies T \implies F \implies$ id, uses every nonterminal symbol in the grammar. This matches with the values in PREV, which gives us each rule number that must be applied to produce the derivation.

3.1.3 Phase 3: Sentence Generation

The third phase of Purdom's algorithm is where the actual generation of sentences occurs. Power and Mallow reformulated the original algorithm by splitting it up into five functions, one main function that contains the sentence generation logic and four auxiliary functions that facilitate the choosing of rules

Stellenbosch University https://scholar.sun.ac.za

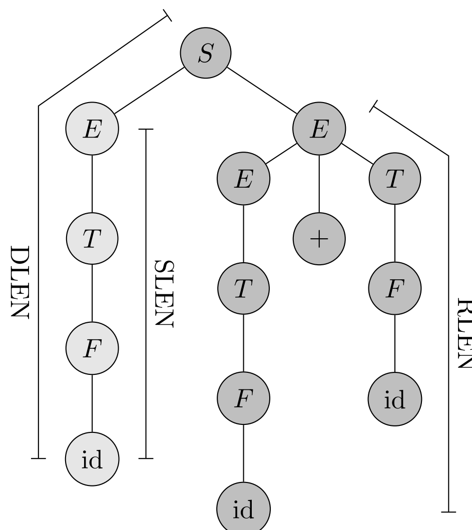


Figure 3.2: Visual example of line 16 in phase 2 of Purdom's algorithm for the production $p = E \rightarrow E{+}T$. Two derivation trees are shown for the derivation of p and the shortest derivation for E. For this production we calculate (DLEN[E] - SLEN[E]) + RLEN[p]. This can be thought of removing the tree corresponding to SLEN[E] and replacing it with the tree corresponding to RLEN[p]. The darker shaded part of the tree shows the result of the calculation.

DLEN	
S	5
E	5
T	5
F	5

PREV	
S	-1
E	1
T	3
F	5

Table 3.2: DLEN and PREV after the completion of the second phase of Purdom's algorithm

and output of sentences. The implementation of Purdom's algorithm adheres to this reformulation with minor modifications to fully adhere to Purdom's original description of this phase.

This phase introduces three new data structures with the purpose of keeping track of the sentences which are being generated, which production have been used and how the productions are to be used. They are:

STACK This data structure maintains the sentence generated as a sentential form. It is implemented as a stack.

ONST This is a list that keeps track how many times each nonterminal occurs in the stack

ONCE This is a list that keeps, for each nonterminal, the current state of the nonterminal. The values in this list fall into the integer range $[-4, |N_G|)$. When a value more or equal to zero is assigned to an index in ONCE it indicates the number of the last production that was used to rewrite

Stellenbosch University https://scholar.sun.ac.za

the nonterminal in a production. The negative values have the following meanings:

-1 **NULL**: The initial value of the production number to be used at the start of the phase.

-2 **READY**: The nonterminal has already been used with the production number that was previously in ONCE and it is now ready to be used again.

-3 **UNSURE**: The value of ONCE that was computed in the previous iteration is unclear.

-4 **FINISHED**: All possible rules for this nonterminal have been exhausted. If it is encountered again it will be expanded to its shortest derivation.

MARK This is a list that, for each rule, keeps track of whether that rule has been used or not.

Auxiliary Functions

As with the previous two phases, the new data structures must be initialised before they can be used. For nonterminal $n \in N_G$, ONCE in initialised to READY and ONST to zero, this indicates that every nonterminal is available for use in a production and that there are no symbols in STACK. For every production $p \in P_G$, MARK is initialised to **False** to indicate that no productions have been used. Algorithm 3.3 shows this initialisation and also shows the other auxiliary methods. They are:

SHORT This function is responsible for returning the production that must be used to yield the shortest string for the nonterminal nt using the SHORT data structure calculated in phase 1 of the algorithm. The value MARK[nt] is changed to **True** to indicate that this production has been used. The value of the production in ONCE is also examined and if this value is not FINISHED, it is set to READY. This modification to Purdom's original algorithm was made by Power and Malloy to indicate that if all possible rules for a symbol nt have not been used, it may be useful in another rule and should be made available.

LOAD ONCE This function prepares ONCE for the next rewriting of nonterminals onto the stack. Each production in the grammar is checked for two conditions, that it has not been used and that is in the READY or UNSURE state. This means that the production is available for use in the sentence generation process. Once this has been determined the number of the production is loaded into ONCE and its index in MARK is set to **True** to indicate that it is to be used. In the reformulation by Power and Malloy only the condition in line 18 is checked. This deviates from Purdom's original description of the algorithm.

PROCESS STACK This function manages STACK in the event that a new symbol needs to be added to or removed from it. The function does this by first decrementing ONST[nt] to indicate that this symbol was removed from the stack. Then all the symbols in the body of *prod_no* are pushed on

Stellenbosch University https://scholar.sun.ac.za

to STACK and if the symbol being pushed is a nonterminal its value in ONST is incremented.

After this replacement of a nonterminal with its corresponding rule the function process the symbols that are in STACK. Symbols are popped from the stack until a nonterminal is encountered. This nonterminal will be the next symbol for which a rule must be chosen. The terminal symbols that are popped before it may be handled by any method. Here it has been decided to print the symbols, but in the implementation they are used to build the sentences generated by the algorithm. In the event that STACK is empty we know that we know that all rules have been exhausted and we may terminate, so `do_sentence` is set to **False**.

It is useful to note here that because a stack is used and terminal symbols are handled as they are popped from the back of the sentential form, if one were to just print the terminal symbols as they are popped off the stack, the sentence would be generated in reverse. This must be taken into account when using the algorithm.

Main Loop

The main loop of the program, shown in algorithm 3.4, is at the core of Purdom's algorithm. It works together with the auxiliary functions to perform the final generation of test sentences.

This phase does not introduce any new data structures, but makes use of those data structures calculated in the two previous phases. When examining algorithm 3.4 it can be broken down as follows:

Lines 4 – 52 This **while** loop guarded by the variable *done* determines when the algorithm has completed the generation of all sentences. This is determined by the if statement on line 11, determining whether all possible rules for the start symbol have been completed, at which point `done` is assign the value **True**.

Lines 9 – 49 This loop controls the generation of a single sentence. For each iteration of this loop a single symbol replaced is with an appropriate production rule. It does so by performing a number of checks before the stack is managed. This may be broken up into a few sections.

Lines 11 – 18 The series of **if-else** statements are evaluated before the production of a sentence is attempted.

1. If all the rules for the start symbol have been exhausted. It is not necessary to generate a sentence and the algorithm can be terminated.
2. If the nonterminal to be used is not the start symbol and all its production rules have been exhausted. The production rule with the shortest derivation is used.
3. If ONCE indicates that the nonterminal is to be used in the next rewrite of a production rule, the production rule to be used is set and the nonterminal is marked READY in ONCE.

Stellenbosch University https://scholar.sun.ac.za

Lines 20 – 49 This final branch of the **if-else** is used to generate a new path from the start symbol that uses only productions that are unused or that lead to the shortest derivation. The PREV data structure is extensively used to build this path. After the path has been created the three checks that were performed are performed again to update the production number to be pushed to the stack for this path.

Line 50 Once the production number has been determined the stack is processed. Each element is pushed onto the stack and ONST is incremented for each pushed to keep track of how many times that symbol appears on the stack.

All the terminal symbols at the top of the stack are then popped off. How these symbols are handled is up to the developer. In this example the symbols are simply printed.

Stellenbosch University https://scholar.sun.ac.za

Algorithm 3.3 Purdom's Algorithm: Phase 3 Auxiliary Methods

```
1: procedure INIT PHASE 3(ONCE, ONST, MARK)
2:     for each nonterminal n ∈ V do
3:         ONCE[n] = READY
4:         ONST[n] = 0
5:     end for
6:     for p ∈ P : MARK[p] = False
7: end procedure
8:
9: procedure SHORT(nt, SHORT, MARK, ONCE)
10:    prod_no = SHORT[nt]
11:    MARK[prod_no] = True
12:    if ONCE[nt] != FINISHED : ONCE[nt] = READY
13:    return prod_no
14: end procedure
15:
16: procedure LOAD ONCE(MARK, ONCE)
17:    for each production p ∈ P do
18:        if MARK[p] == False then
19:            J = LHS[p]
20:            if ONCE[J] == READY or ONCE[J] == UNSURE then
21:                ONCE[J] = p
22:                MARK[p] = True
23:            end if
24:        end if
25:    end for
26: end procedure
27:
28: procedure PROCESS STACK(nt, prod_no, do_sentence, ONST, STACK)
29:    ONST[nt] -= 1
30:    for each element e ∈ RHS[prod_no] do
31:        STACK.push(e)
32:        if e ∈ V : ONST[e] += 1
33:    end for
34:    done = False
35:    while done == False do
36:        if STACK is empty then
37:            do_sentence = False
38:            break
39:        else
40:            nt = STACK.top()
41:            STACK.pop()
42:            if nt ∈ Σ : print(nt)
43:            else : done = True
44:        end if
45:    end while
46: end procedure
```

Stellenbosch University https://scholar.sun.ac.za

Algorithm 3.4 Purdom's Algorithm: Sentence Generation

```
1: procedure SENTENCE GENERATION
2:   done = False
3:   prod_no = Null
4:   while done is False do
5:     if ONCE[S] is FINISHED then
6:       break
7:     end if
8:     ONST[S] = 1 : nt = S : do_sentence = True
9:     while do_sentence is False do
10:      once_nt = ONCE[nt]
11:      if once_nt is FINISHED and nt is S then
12:        done = True
13:        break
14:      else if once_nt is FINISHED then
15:        prod_no = SHORT(NT)
16:      else if once_nt ≥ 0 then
17:        prod_no = once_nt
18:        ONCE[nt] = READY
19:      else
20:        LOAD ONCE()
21:        for each nonterminal I ∈ V do
22:          if I is not S and ONCE[I] ≥ 0 then
23:            J = I
24:            K = PREV[I]
25:            while K ≥ 0 do J = LHS[K]
26:              if ONCE[J] is 0 then
27:                break
28:              else if ONST[I] is 0 then
29:                ONCE[J] = K
30:                MARK[K] = True
31:              else
32:                ONCE[J] = UNSURE
33:              end if
34:              K = PREV[J]
35:            end while
36:          end if
37:        end for
38:        for each nonterminal n ∈ V do
39:          if ONCE[n] is READY : ONCE[n] = FINISHED
40:        end for
41:        if nt is S and ONCE[nt] is FINISHED and ONST[S] is 0 then
42:          break
43:        else if ONCE[nt] < 0 then
44:          prod_no = SHORT(nt)
45:        else if ONCE[nt] >= 0 then
46:          prod_no = ONCE[nt]
47:          ONCE[nt] = READY
48:        end if
49:      end if
50:      PROCESS STACK()
51:    end while
52:  end while
53: end procedure
```

Stellenbosch University https://scholar.sun.ac.za

3.2 Zelenov and Zelenova

The research conducted by Zelenov and Zelenova on the generation of test inputs from a context free grammar focusses on the construction and satisfaction of coverage metrics to ensure that the test input sets generated adequately cover the input of a family of parsers. They focus on a number of coverage metrics applicable to LL and LR parsers, taking into consideration the mechanisms used by these parsers when validating input sentences.

The algorithms proposed here have some advantages over Purdom's method. The algorithms do not require the construction of any special data structures as they are, firstly, described in terms of well-known operations common to context free grammars and, secondly, they are not described iteratively, but rather in terms of sets that must be generated, leaving the ultimate implementation up to the reader. This section will discuss the sets generated to cover the PLL, WPLR, and NLL/NLR coverage metrics.

3.2.1 Preliminaries

Before the methods of construction for the sets to satisfy the different coverage metrics can be given it is necessary to provide algorithms that illustrate how to compute some important sets that facilitate in the generation of the sets.

First Set Construction

For a given context free grammar, $G = (T_G, N_G, P_G, S_G)$, the first set of a grammar symbol given in definition 2.4 must be modified so that it accepts sentential forms so that $\mathit{first}(\alpha) = \{t \in T_G : \alpha \overset{*}{\Longrightarrow} t\beta\}$. This new first set can be constructed using the following rules:

1. If α is a terminal symbol, $first(\alpha) = \alpha$
2. If α is a nonterminal symbol, $first(\alpha) = \bigcup_{\alpha \to \gamma} first(\gamma)$
3. If $\alpha = \alpha_0\alpha_1 \dots \alpha_n$ is a sentential form, the first set of α is:
   (a) $first(\alpha) = first(\alpha_0)$ if the rule $\alpha_0 \to \epsilon$ does not exist
   (b) $first(\alpha) = first(\alpha_0) \cup first(\alpha_1 \dots \alpha_n)$ if the rule $\alpha \to \epsilon$ does exist.
   (c) $first(\alpha) = first(\alpha) \cup \epsilon$ if the rule $\alpha_k \to \epsilon$ exists for all $0 \leq k \leq n$

When constructing sets of sentences, the definition of the first set must be adjusted so that, when constructing the first set of a sentential form, $\alpha = \alpha_0\alpha_1 \dots \alpha_n$, the result is a set of sentential forms of the form $w\beta$, where $w \in T_G$ and $\alpha \overset{*}{\Longrightarrow} w\beta$. This new definition of the first set produces a set of sentential forms that are made up of a terminal, followed by the remainder of the sentential form of the rule from which that symbol was taken, followed by the remainder of α. This allows the expansion of a symbol in a sentential form without introducing any errors in to the sentential form. Algorithm 3.5 illustrates how to construct the first set for this new definition, which will be called $first_n$.

When working with a grammar item, $i = p \to \omega \cdot A\beta$, we define the first for this item in terms of $first_n$ as

$$first_i(i) = \{\omega \cdot \alpha\beta \mid \alpha \in first_n(A)\}$$

Stellenbosch University https://scholar.sun.ac.za

Algorithm 3.5 Construction of $first_n(\alpha)$

```
1:  procedure FIRST(α)
2:      set = ∅
3:      if α ∈ T_G then
4:          return set ∪ α
5:      else if α ∈ N_G then
6:          for p = α → γ ∈ P_G do
7:              set = set ∪ FIRST(γ)
8:          end for
9:          return set
10:     else if α ∈ (T_G ∪ N_G)* then
11:         for α_k ∈ α_0 ... α_n do
12:             set = set ∪ {wα_{k+1} ... α_n | w ∈ FIRST(α_k)}
13:             if α_k → ϵ does not exist: break
14:         end for
15:         if α_k → ϵ for all α_k ∈ α: set = set ∪ ϵ
16:         return set
17:     end if
18: end procedure
```

Derivation Chain Construction

For a given context free grammar, $G = \{T_G, N_G, P_G, S_G\}$, the derivation chain for a symbol, x, may be constructed via a breadth first search of the production rules as follows:

1. Let x be the vertex a directed graph $H = (\{x\}, \emptyset)$
2. Construct the set of nonterminals, D_x, such that x is a successor of every $t \in D_x$. For every $t \in D_x$, add to H the edge (t, x).
3. Step 2 is repeated until S_G is a vertex in H. At which point the desired chain has been computed.

Algorithm 3.6 Construction of a derivation chain

```
1: procedure DERIVATION CHAIN(x ∈ (T_G ∪ N_G))
2:     H = (F = {x}, E = ∅)
3:     while S ∉ N_G do
4:         for every leaf, l, of H do
5:             D = {t | t → αlβ ∈ P_G}
6:             for t ∈ D: F = F ∪ {t} and E = E ∪ {(t, l)}
7:         end for
8:     end while
9: end procedure
```

At the completion of the steps above, a directed graph with a path from S, to the desired symbol will have been constructed. To construct the derivation it is only need to follow the path constructed from S to x. As the search conducted

Stellenbosch University https://scholar.sun.ac.za

is done in the manner of a breadth first search it is guaranteed that the shortest derivation chain was found. Algorithm 3.6 illustrates the above steps.

3.2.2 PLL

Positive LL coverage described in definition 2.11 requires a set of test inputs such that every nonterminal must be derived to a sentential form that starts with each terminal in its first set. If the sentential form $s = \alpha A \beta$ is available we can construct the set, W_{PLL_A}, of sentential form such that

$$W_{PLL_A} = \{\alpha x \omega \beta \mid x \in first(A) \text{ and } x\omega \in first_n(A)\}$$

To satisfy full PLL we must create a set W_{PLL} that is the union of all the sets W_{PPL_N} for every nonterminal N in the grammar.

$$W_{PLL} = \bigcup_{N \in N_G} W_{PLL_N}$$

The satisfaction of PLL coverage, from the construction of the sentential forms to a set of terminal string that satisfy the metric, is a four step process. For each nonterminal, A, in the grammar we perform the following steps:

1. Build the shortest derivation chain from A to S_G
2. Use the derivation chain to construct the sentential form $\alpha A \beta$
3. Replace each A in $\alpha A \beta$ with a member of $first_n(A)$, to give us W_{PLL_A}.
4. For each member of W_{PLL_A}, replace all the nonterminal symbols with a string of appropriate terminal symbols.

The process of constructing a sentential form from a derivation chain will be described here. Constructing a sentence in the grammar from the sentential form will be described in Section 5.2 as it is more general step applicable to all the methods of test generation.

Construct a Sentential Form

For any nonterminal, A, in a grammar, $G = (T_G, N_G, P_G, S_G)$, it has been described how to construct the shortest derivation chain, $A < B_0 < \cdots < B_k < S_G$, from A to S_G. To construct a sentential form from this derivation chain we iterate through the derivation chain in reverse order and, starting with the start symbol, create a set of expanded sentential forms where each sentential form is created by inserting a occurrence, with respect to the start symbol, into the original sentential form until the end of the chain of the chain is reached. This gives us a set, for any nonterminal symbol, A, of sentential forms. This method is given in algorithm 3.7.

$$D_A = \{\alpha A \beta \mid S_G \overset{B_k}{\Longrightarrow} \cdots \overset{B_0}{\Longrightarrow} \alpha A \beta\}$$

As an illustration consider constructing the set of sentential forms for F in the example grammar. For this construction the derivation chain $F < T < E < S$ is constructed. Starting with the S, we create a new sentential form by replacing

Stellenbosch University https://scholar.sun.ac.za

every occurrence of E in a rule $S \to \alpha E \beta$ with a rule $E \to \alpha T \beta$. This gives yields the sentential forms E and $E+T$. If this is continued until the F is added the set $\{E+F, E+T*F, T*F, F\}$ is constructed, at which the generation of sentential forms is complete. Figure 3.3 gives a visual representation of this process.

Algorithm 3.7 Constructing a sentential form from a derivation chain

1: **procedure** CONSTRUCT SENTENTIAL FORM(A : chain from A to S)
2: chain = $A < B_1 < \cdots < B_{n-1} < B_n < S_G$ ▷ $S = B_{n+1}$
3: sf = $\{S_G\}$
4: **for** each symbol B_k **in** chain **where** $k = n \ldots 0$ **do**
5: b = $\{\alpha B_k \beta \mid B_{k+1} \to \alpha B_k \beta\}$
6: newsf = $\emptyset$
7: **for** each s = $\gamma B_{k+1} \omega$ **in** sf **do**
8: newsf = newsf $\cup$ $\{\gamma \alpha B_k \beta \omega\}$
9: **end for**
10: sf = newsf
11: **end for**
12: **end procedure**

Once the sentential forms have been constructed one may be chosen to be used as for the coverage of PLL$_F$. To minimise the size of the test inputs elected to choose the shortest sentential form in the set, F.

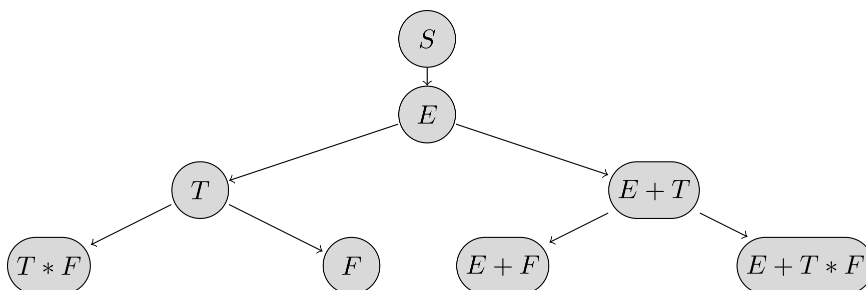


Figure 3.3: Construction of set of sentential forms for derivation chain for F to S

The selection of the sentential form F results in the test inputs 'id' and '(', each covering the pairs $(F,\ id)$ and $(F,\ ()$. More test inputs are needed to fully satisfy PLL with the set $\{id, (id), id{+}id, id{*}id\}$ achieving this. With id covering (E, id), (T, id), (F, id) and (id, id) and (covering $(E, ()$, $(F, ()$, $(T, ()$, $(F, ()$, (id) covering $((, ()$, $id + id$ covering $(+, +)$ and $(id * id)$ covering $(*, *)$.

3.2.3 WPLR

Weak positive LR (WPLR) coverage, described in definition 2.12 requires a test set such that for every item, every terminal in the first set for a grammar symbol after the dot in the item is used in the derivation of the grammar symbol at

Stellenbosch University https://scholar.sun.ac.za

the head of the item. This is very similar to PLL coverage except that here the coverage metric is defined over items in the grammar and not symbols. This is more robust as more sentential forms in the grammar in the grammar must be covered to satisfy WPLR coverage. With the example grammar PLL, as shown earlier, the set $\{id, (id), id + id, id * id\}$ satisfies PLL coverage, but this set does not include sentential forms such '$(E) + F$' or '$((E))$'. This is not sufficient when testing an LR parser as the sentential are needed to explore as many paths as possible in the state machine. WPLR, being defined over all the items of a grammar, guarantees that all the states in the automaton will be explored as the set of kernel items in each state is unique.

The satisfaction of WPLR, for a item $\pi = N \rightarrow \alpha \cdot A\beta$, is a similar process to that of PLL coverage with the following steps for all items in the grammar:

1. Build a derivation chain from A to S_G
2. Use the derivation chain to construct a sentential form $\alpha A\beta$
3. Replace each A in $\pi = N \rightarrow \alpha \cdot A\beta$ with a member of $first_n(A)$ to get W_{WPLR_A}
4. Replace all the nonterminal symbols in each member of W_{WPLR_A} with the appropriate terminal symbols to produce a sentence.

For the example we have the following kernel items:

$$
\begin{array}{ll}
E \rightarrow \cdot E + T & E \rightarrow E \cdot +T \\
E \rightarrow E + \cdot T & E \rightarrow \cdot T \\
T \rightarrow \cdot T * F & T \rightarrow T \cdot *F \\
T \rightarrow T * \cdot F & T \rightarrow F \\
F \rightarrow id & F \rightarrow \cdot (E) \\
F \rightarrow (E \cdot) &
\end{array}
$$

This test set will yield may more test inputs covering a much larger part of the input space than PLL coverage. This does not, however, contain the item $F \rightarrow (\cdot E)$ which will result in the coverage of all transitions, satisfying PLR coverage.

3.2.4 NLL

NLL (Negative LL) coverage requires that every token, A, in the grammar be preceded by some token in R_A. To do this we must know what tokens can not come before A. We have shown how R_A may be constructed, but here we follow a variation of the method described in [23].

First, given a grammar G, we construct a new grammar, H, which is the same as G, but with the right hand side of the rules reverse, when we consider these as strings. In terms of our example grammar H will be

$$
\begin{aligned}
&S \rightarrow E \\
&E \rightarrow T + E \mid T \\
&T \rightarrow F * T \mid F \\
&F \rightarrow)E(\mid \mathbf{id}
\end{aligned}
$$

Stellenbosch University https://scholar.sun.ac.za

Once we have constructed H, we can construct the set of negative test inputs in H by the following process. For any symbol, x, in $T_H \cup N_H$, we construct $follow(x)$. We construct F_x as the compliment of the follow set of x. We now build a sentential form, $\alpha x \beta$, for x using the derivation chain as described above. We construct the set of mutated sentential forms by inserting each symbol in F_x behind x in the sentential form so that we have

$$N_H = \{\alpha x t \beta |\ t \in F_x\}$$

Each sentential form in N_H can now be reversed to construct sentential forms that may be transformed into sentences in G through the replacement of all nonterminal symbols with appropriate sentential forms.

As an example let us construct a set of negative sentential forms for the symbol E. In the grammar H $follow(E) = \{(\}$. We take the compliment to get the set, $\{\mathbf{id}, +, *, (\}$, of symbols that cannot follow E in H. We can use the sentential form $)E($ and construct the set of invalid sentential forms in H.

$$N_H = \{)E\mathbf{id}(,\ )E)(,\ )E * (,\ )E + (\}$$

The sentential forms in N can now be reversed to form a set of invalid sentential forms in G

$$N_G = \{(\mathbf{id}E),\ ()E),\ (*E),\ (+E)\}$$

This process must be repeated for every symbol in G to satisfy NLL coverage.

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Chapter 4

LR-Automata Search

As discussed in section 2.3, the operation of an LR parser may be described through the use of a state stack, symbol stack, input buffer and a finite state machine. The finite state machine assists in the visualisation of how the parser performs the shift, reduce and goto operations.

The FSM may be represented by a directed graph that can be exploited to generate test inputs through the applicatoin of generic search algorithms, such as breadth first search. This chapter will present two methods which allow us to construct strings for an LR grammar.

Definition 4.1. *A parsing automaton, P, for the grammar, $G = (T_G, N_G, P_G, S_G)$, is defined by the 5-tuple $P(G) = (T_P, S, \delta, s_0, F)$ where:*

1. *$T_P = T_G \cup N_G \cup \{\$\}$ is a finite set representing all the symbols in G. The end-of-input symbol \$, is a symbol not in T_G or N_G..*
2. *$S = C \cup \{s_e\}$ is a finite, non-empty, set where C is the canonical collection of LR(k) item sets and s_e is the error state. This represents the states in the state machine.*
3. *δ is a function from $S \times T_P$ to S. This function represents the transitions in P.*
4. *$s_s \in S$ is the start state for this machine.*
5. *$F \subseteq S$ is a finite set representing the accept states for this machine.*

Figure 4.1 shows the parsing automaton for the example grammar. This may be used to construct the parsing table shown in table 4.1. Each row in the table represents a state in the parsing automaton, the number for each row, $1 \ldots 11$, corresponds to states $S_0 \ldots S_{11}$ in the parsing automaton. The columns in the Action portion of the table indicate the action that must be taken in each state depending on what the next input token is. A shift from state i to state j when the next token is x is represented by the entry, s_j, in row i and column x. A reduction using rule number p is indicated by the entry r_p. The Goto portion of the table indicates which state to transition to after a reduction has been performed depending on what token is on top of the symbol stack. An entry, j, in row s_i and column A indicates that after a reduction, if state s_i as at the

Stellenbosch University https://scholar.sun.ac.za

top of the state stack and token A is at the top of the symbol stack the parser must push state s_j onto the state stack.

With this definition of the parsing automaton we can now define the structure of the sentences that the normal shift, reduction and goto operations performed by and LR parser may be applied to.

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r7		r7	r7			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Table 4.1: Parse Table for running example 2.1

Definition 4.2. *A test state, t, for a parsing automaton, P, is defined as the 3-tuple $t = (s, \omega, \alpha)$ where:*

1. *$s = s_0 \ldots s_n$ is finite, non empty, list representing the stack of states for t, where every member of s is also a member of S.*
2. *$\omega = \omega_0 \ldots \omega_m$ is a string representing the sentence for t, where every symbol in ω is in T_G.*
3. *$\alpha = \alpha_0 \ldots \alpha_q$ is a string representing the current right most derivation for t, where every symbol in α is in $T_G \cup N_G$.*

A test state is valid for a parsing automaton, P, if s corresponds to a valid parser configuration, that is, for a test state, $t = (s, \omega, \alpha)$, there exists a parser configuration, $s_0 \ldots s_n\theta$, such that $s_0 \ldots s_n = s$. The test state $t = (s_s, \epsilon, \epsilon)$, is valid for all parsing automaton and is used as the starting test case for the methods presented in this chapter. This test state will be refered to as the trivial test state.

Shift and Goto Operations

Every test state that is valid for a parser may be shiftable, or reducible. This means that the test state corresponds to a parser configuration from which a shift or reduction can be performed. It was possible for a test state to be both shiftable and reducible. An example of a test state that is both shiftable and reducible in the context of the automaton constructed for the example grammar

Stellenbosch University https://scholar.sun.ac.za

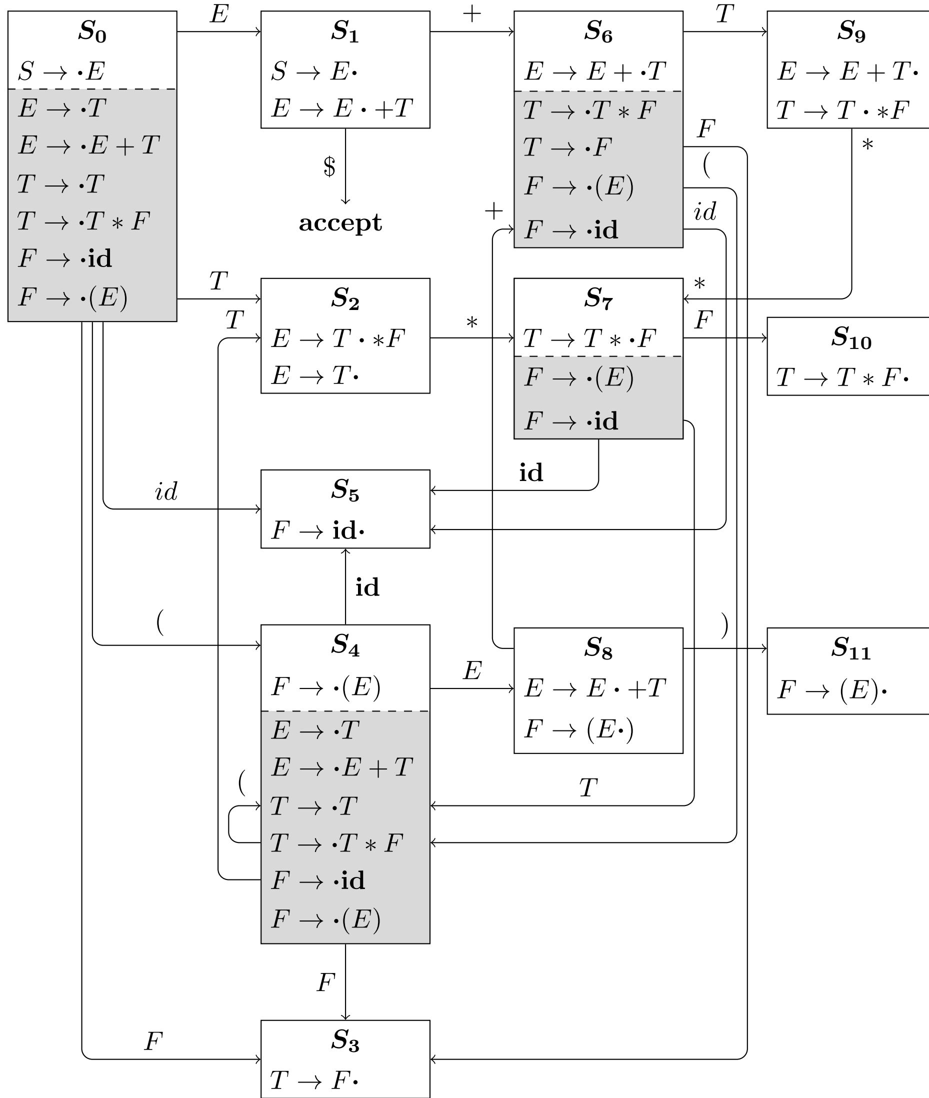


Figure 4.1: Parsing automaton for running example2.1

is the test state $([0, 2], \mathbf{id}, T)$. This test state reducible by the rule $E \to T$ and shiftable on the token $*$.

Definition 4.3. *A test state, $t = (s_0 \dots s_n, \omega, \alpha)$, is shiftable for a parser if there exists a $x \in T_G$ such that $\delta(s_n, x) \neq s_e$.*

Definition 4.4. *A test state, $t = (s_0 \dots s_n, \omega, \alpha_0 \dots \alpha_q)$, is reducible for a parser if s_n contains any item of the form $A \to \alpha_i \dots \alpha_q \cdot$.*

Now the shift, reduction and goto operations performed may be defined. These operations produce the same results as those of an LR parser. The shift operation pushes the next state on to s, and pushes the terminal symbol to ω

Stellenbosch University https://scholar.sun.ac.za

and α. The reduction operation, when reducing by a rule $A \rightarrow \alpha$, pops the $|\alpha|$ symbols off s and α. The goto operation pushes the next state to s, and appends the nonterminal symbol to α.

Definition 4.5. *Let $t = (s_0 \dots s_n, \omega, \alpha)$ be a shiftable test state and let $k \in T_G$. The function $s(t, k)$ defines the shift operation on t.*

$$s(t, k) = (s_0 \dots s_n \delta(s_n, k),\ \omega k,\ \alpha k)$$

If t is not a shiftable test case $s(t, k) = t$.

Definition 4.6. *Let $t = (s, \omega, \alpha)$ be a reducible test state and let $A \rightarrow \alpha\cdot$ be an item in s_n. The function r(t, p) defines the reduction operation on t.*

$$r(t, p) = (s_0 \dots s_{n-|\beta|}, \omega, \alpha_0 \dots \alpha_{q-|\beta|})$$

If t is not reducible $r(t, p) = t$

Definition 4.7. *Let t be a reducible test state and let $p \rightarrow \beta\cdot$ be an item s_n. If $t_r = r(t, p) = (s, \omega, \alpha)$ then the function $g(t_r, p)$ defines the goto operation on t_r*

$$g(t_r, p) = (s\ \delta(s_n, p), \omega, \alpha p)$$

Definition 4.6 may be rewritten as $r_g = g(r(t, p), p)$ to define the reduction and incorporate the goto transition performed directly after it. With this and definition 4.5 we have all the operations an LR parser may perform.

When constructing test states for a grammar, G, we define a complete test state, that is, a test state where ω is a sentence in the language generated by $L(G)$, as any test state $t = (s, \omega, S_T)$.

4.1 Breadth First Search

The approach for searching a LR-automaton is quite simple. We may apply any generic search algorithm to the automaton to generate any number of test states. Here a simple method for doing so using breadth first search is given.

For any given shiftable test state there may be a number of shift transitions in the automaton that may be applied to it. The application of all possible shift transitions to a shiftable test state is defined as

$$s_{bfs}(t) = \{s(t, k) | \delta(s_n, k) \neq s_e \text{ and } k \in T_G\}$$

This is the core of the algorithm as it tells us which transitions to add to the search frontier. The algorithm is presented in algorithm 4.1. While this algorithm allows us to any number of strings accepted by the parser it suffers when implemented in practice for two reasons:

Time explosion The vast majority of context free grammars found in practice produce languages of infinite size. As the frontier is expanded by the average branching factor in the FSM for each new token that is added to its elements, the running time of the algorithm is exponential.

Stellenbosch University https://scholar.sun.ac.za

Cycles in the FSM Any cycles in the FSM, are continuously searched with each iteration, this adds a significant number of test cases to the frontier that will never be completed.

Algorithm 4.1 Breadth First Search Over a Finite State Machine

```
procedure FINITE STATE MACHINE BFS
    f ← [(s_0, ϵ, ϵ)]
    while |f| > 0 do
        p ← f.pop()
        if p is shiftable then
            f = f ∪ s_bfs(p)
        else if p is reducible then
            f = f ∪ {r_g(p)}
        else if p is a complete test case then
            yield p
        end if
    end while
end procedure
```

4.2 Breadth First Search with Termination

While generic breadth first search allows us to generate any number of valid test cases for a finite state machine, in practice we want an algorithm that is guaranteed to terminate and generate test cases that satisfy specific criteria. To achieve this we extend the algorithm to include search termination conditions. This algorithm attempts to satisfy PLR coverage through an explicit search of the LR-automaton. While PLR is coverage is not guaranteed to be satisfied for any given automaton we see in chapter 6 that the test inputs produced by it correlate best to manually constructed test states when used as a means of assessing the quality of parsers. The method may be split into two distinct parts excluding the replacement of all nonterminal symbols with an appropriate string of terminal tokens.

1. The finite state automaton is searched using any search method and termination conditions which will yield a set of incomplete test cases.
2. Each test case is completed using the kernel items to produce a valid sentential form.

4.2.1 Search

The method of termination will determine the coverage that is achieved when conducting a search on the FSM. We associate with each state in the automaton from which a shift operation may be performed a boolean value that indicates whether or not the state has been previously accessed in the search. All such states are initially associated with a false value. We expand shiftable test state by means of a breadth first search, marking each state in the FSM that has

Stellenbosch University https://scholar.sun.ac.za

been visited as true. When we see that a shiftable test state been expanded into a state that has been previously marked we remove that test state from the search frontier and add to a set of test states that must be completed by the second step. There will always only be one test state that is completed by end of the search. Algorithm 4.2 shows how this step may be performed.

Algorithm 4.2 Breadth First Search over a FSM with termination conditions

1: **procedure** BFS WITH TERMINATION
2: $f = [(s_0,\ \epsilon,\ \epsilon)]$
3: $state = [i : \textbf{False for } s_i \textbf{ in } S]$
4: $test_cases = \emptyset$
5: **while** $|f| > 0$ **do**
6: $p = (s_0 \ldots s_p, \omega, \alpha) = f.pop()$
7: **if** p **is** shiftable **and** $state[s_p] =$ **False then**
8: $state[n] =$ **True** : **for** $(s_0 \ldots s_n, \omega, \alpha)$ **in** $s_{bfs}(p)$
9: $f = f \cup s_{bfs}(p)$
10: **else if** p **is** shiftable **then**
11: $test_cases = test_cases \cup \{p\}$
12: **end if**
13: **if** p **is** reducible **then**
14: $f = f \cup\ r_g(p)$
15: **end if**
16: **end while**
17: **end procedure**

4.2.2 Completion of Test Cases

After the initial search has been completed each test state in the *test_cases* will be incomplete. To complete complete these test state we continue the search in the finite state machine, but instead of appending only tokens from the terminal transitions to ω, we append the tokens associated with the transitions attached to the kernel items in each state. These transitions are followed and the appropriate reductions are performed until the test state is complete. This is shown in Algorithm 4.3

The method of following the kernel item transitions for a test state is, with the correct choice of kernel items, guaranteed to complete. All kernel items, except the item associated with the starting symbol, are of the form $p \to \theta \cdot \gamma$, where $\theta \neq \epsilon$, this means that upon the reduction of a kernel item, we always pop more items of the symbol and state stacks that we had to push to them to complete the rule.

It must be noted that the choice of the kernel item is important in guaranteeing the completion of the algorithm. It is possible to choose the kernel items in a such a way that the completion of the procedure cannot be guaranteed as recursive rules form loops in the finite state machine. Figure 4.2 illustrates a situation, with respect to the example grammar and its finite state machine, where if one were to randomly choose a kernel item after each shift or reduction it is possible to perform multiple shifts and reductions and to be left with the same sentential form.

Stellenbosch University https://scholar.sun.ac.za

Algorithm 4.3 Completion of test cases after breadth first search

```
procedure COMPLETE TEST CASE(t : test case)
    while t = (s_0 ... s_n, ω, α) is not complete do
        k = A → γ · β                                   ▷ kernel item from s_n
        if β is ε then
            t = r_g(t, k)
        else
            x_0 ... x_n = β
            t = (s_0 ... s_n δ(s_n, x_0), ωx_0, αx_0)
        end if
    end while
    return t
end procedure
```

State	Symbols	Action	Kernel Item
Here $F \rightarrow E \cdot +T$ is randomly chosen as the kernel item			
$S_0\ S_4\ S_8$	$(\ E$	Shift $+$	$E \rightarrow E \cdot +T$
$S_0\ S_4\ S_8\ S_6$	$(\ E+$	Shift T	$E \rightarrow E + \cdot T$
$S_0\ S_4\ S_8\ S_6\ S_9$	$(\ E\ +\ T$	Reduce by $E \rightarrow E + T$	$E \rightarrow E + T \cdot$
Here $F \rightarrow E \cdot +T$ is randomly chosen as the kernel item			
$S_0\ S_4$	$(\ E$	Reduce by $E \rightarrow E + T$	$E \rightarrow E\ \cdot + T$
	⋮		

Figure 4.2: Infinite loop during the completion of a test case

After the complete of algorithm 4.3 we are left with a test case where ω is a valid sentential form in G. As with the other algorithms presented these sentential are completed by replacing all the nonterminal symbols with an appropriate string of terminal strings.

Negative Test Case Generation

To generate a set of negative test states we must make a small modification to Algorithm 4.2. We search the LR-automaton as if we were constructing positive test states but, before we add each expanded test state to the search frontier, we construct a set of test states where an invalid token has been added. This can be done by collecting all the test cases constructed by shifts on tokens that lead us to the error state and adding the test state to the set of incomplete, but not appending the tokens, or the new state to the sentential form or state stack of the test state so that it may be reduced correctly. Algorithm 4.4 shows the modification made to the search algorithm to construct a set of incomplete negative test states.

Stellenbosch University https://scholar.sun.ac.za

Algorithm 4.4 Breadth First Search Generating Negative Test States

```
 1: procedure BFS WITH TERMINATION
 2:     f = [(s_0, ϵ, ϵ)]
 3:     state = [i : False for s_i in S]
 4:     test_cases = ∅
 5:     test_cases_neg = ∅
 6:     while |f| > 0 do
 7:         p = (s_0 ... s_p, ω, α) = f.pop()
 8:         if p is shiftable and states[s_p] = False then
 9:             state[n] = True : for (s_0 ... s_n, ω, α) in s_bfs(p)
10:             t_n = {(s_0 ... s_p, ωδ(s_p, k), α) : δ(s_p, k) = s_e}
11:             test_cases_neg = test_cases_neg ∪ t_n
12:             f = f ∪ s_bfs(p)
13:         else if p is shiftable then
14:             test_cases = test_cases ∪ {p}
15:         end if
16:         if p is reducible then
17:             f = f ∪ r_g(p)
18:         end if
19:     end while
20: end procedure
```

Stellenbosch University https://scholar.sun.ac.za

Chapter 5

Tool Implementation

In this chapter we provide a overview of the implementation of the test case generation framework. Along with the program structure, here we will also provide usage instructions and descriptions of the intermediary algorithms and structures that were used to transform the input data.

Figure 5.1 shows the structure of the framework, which will be referred to in this Chapter as the generator, implemented to satisfy the coverage criteria previously discussed. Figure 5.1 also shows how information flows thought the generator, from left to right, in the following steps:

Input The input file, specifying the CFG is read and transformed to an internal EBNF representation of the grammar.

Conversion This EBNF is then expanded to BNF and transformed to the internal representation used for CFGs.

Generation The CFG may be used to generate an appropriate parse table or this parse table may be read from a *hyacc* file.

Output The algorithms may then be applied by to the CFG or parse table to produce the output files which contain the test cases

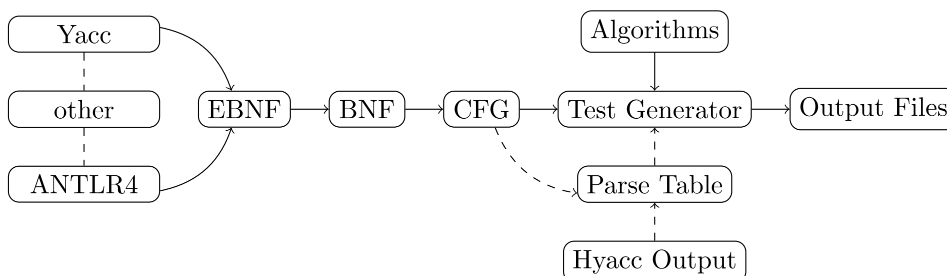


Figure 5.1: Program structure

Stellenbosch University https://scholar.sun.ac.za

5.1 Conversion From Input to BNF

To allow the generator to accept multiple formats for the specification of a CFG, it uses its own internal representation of a CFG. A CFG read from an input file must be converted to this format. To do this, the input must first be converted from its original format to an EBNF description of the grammar, which is expanded to a BNF description and ultimately transformed to a CFG description.

5.1.1 Input to EBNF

The ANTLR 4 [21] framework was was used to construct the two parsers to read the input grammars and convert them to EBNF format. One parser was constructed for ANTLR files, and another for grammars described in the Yacc format. Once the input has been parsed a parse tree is contracted. The parse tree is traversed by a parse tree walker and used to construct the EBNF description of the grammar.

The EBNF representation used by our framework is only necessary when parsing an ANTLR grammar as all Yacc grammars must be specified in BNF format. The EBNF representation used in the framework incorporates some extensions to the BNF description of grammars that may be described in an ANTLR grammar. They are:

Optionals A grammar symbol in a rule followed by a question mark (?) is considered optional to the rule. In the rule $A \rightarrow$ **bc**?**d**, the symbol **c** does not have to appear and A may be derived to **bd**.

Zero-or-more A grammar symbol in a rule followed by a star (*) may appear consecutively any number of times, including zero. In the rule $A \rightarrow$ **bc*d**, A may be derived to **bd** or **bc** $\cdots$ **cd**.

One-or-more A grammar symbol in a rule followed by a plus(+) must appear at least once and may appear consecutively any number of times more than one. In the rule $A \rightarrow$ **bc*d**, A must be derived to **bcd** or **bc** $\cdots$ **cd**.

Groupings The optional, zero-or-more or one-or-more symbols may be applied to a sequence of symbols that are enclosed in parentheses ('(', ')'). The rule $A \rightarrow$ (**abc**)+ derives **abc**, **abcabc** or **abc ... abc**.

Alternations (|) may also be used in a grouping. The rule $A \rightarrow$ (**a** | **b** | **c**) may be derived to **a**, **b** or **c**.

The EBNF is represented as a list of rules. Each rule is represented as a list of expressions where an expression may be a nonterminal or terminal in the grammar, or any of the allowable EBNF operations, namely, the optional, zero-or-more, one-or-more or grouping operations. Consider the grammar in figure 5.2. The *write* and *program* rules in the grammar would be represented as shown in figure 5.3. Here we see how each rule is structures as a list of nested expressions. The only expressions that may contain more than one nested expression are the root expression that holds the entire production rule and an expression that represents a grouping. This EBNF representation of the grammar is what must be converted to a BNF description before the algorithms are applied to it.

Stellenbosch University https://scholar.sun.ac.za

$$\begin{aligned}
\text{program} &\rightarrow \text{BEGIN stmt} + \text{ END} \\
\text{stmt} &\rightarrow \text{assign} \mid \text{write} \\
\text{assign} &\rightarrow \text{IDENT} := \text{EXPR} \\
\text{write} &\rightarrow \text{WRITE EXPR (, EXPR)}*
\end{aligned}$$

Figure 5.2: CFG to illustrate the structure of the EBNF grammar representation

```
production: write
expression:
  expression: terminal
    WRITE
  expression: terminal
    EXPR
  expression: zero-or-more
    expression: grouping
      expression: terminal
        ,
      expression: terminal
        EXPR
```

```
production: program
expression:
  expression: terminal
    BEGIN
  expression: one-or-more
    expression: nonterminal
      stmt
  expression: terminal
    END
```

Figure 5.3: EBNF representation of the *write* and *program* rules

5.1.2 EBFN to BNF

The conversion of a grammar represented in EBNF to one that is represented in BNF is the process of expanding the grammar to remove all EBNF operations that may be present. Here we present how to do this.

When a rule is expanded into multiple rules to remove an EBNF operation the new rules that created are called p_i, where i is a number, starting at 0 and incremented for every new rule that is added.

Optional Expansion For a rule $A \rightarrow$ a? we create two new rules $p_0 \rightarrow$ a and $p_0 \rightarrow \epsilon$. We then change the rule $A \rightarrow$ a? to $A \rightarrow p_0$.

Zero-or-more Expansion For a rule $A \rightarrow$ a$*$ we create two new rules $p_0 \rightarrow \epsilon$ and $p_0 \rightarrow p_0$ a. The rule A is then changed to $A \rightarrow p_0$.

One-or-more Expansion For a rule $A \rightarrow$ a$*$ we create two new rules $p_0 \rightarrow$ a and $p_0 \rightarrow p_0$ a. The rule A is then changed to $A \rightarrow p_0$.

Grouping Expansion For a rule $A \rightarrow (a_1 \mid \cdots \mid a_n)$ we first change the rule to $A \rightarrow p_0$ then we create n new rules, $p_0 \rightarrow a_i$, for $1 \leq i \leq n$.

As an example we illustrate how the grammar in figure 5.2 may be expanded to a grammar in BNF form through the expansion of the *program* and *write* rules p_0.

The rule program $\rightarrow$ BEGIN stmt+ END is expanded by creating two new rules for the expression stmt+, $\mathrm{p}_0 \rightarrow$ stmt and $\mathrm{p}_0 \rightarrow \mathrm{p}_0 stmt$. The program rule is then changed to program $\rightarrow$ BEGIN p_0 END.

Stellenbosch University https://scholar.sun.ac.za

$$
\begin{aligned}
\text{program} &\rightarrow \text{BEGIN } \text{p}_0 \text{ END}\\
\text{p}_0 &\rightarrow \text{stmt}\\
\text{p}_0 &\rightarrow \text{p}_0 \text{ stmt}\\
\text{stmt} &\rightarrow \text{assign} \mid \text{write}\\
\text{assign} &\rightarrow \text{IDENT} := \text{EXPR}\\
\text{write} &\rightarrow \text{WRITE EXPR } \text{p}_1\\
\text{p}_1 &\rightarrow \epsilon\\
\text{p}_1 &\rightarrow \text{p}_1 \text{ , EXPR}
\end{aligned}
$$

Figure 5.4: Grammar after the all the rules have been expanded

The rule write $\rightarrow$ WRITE EXPR (, EXPR)$*$ is expanded by first expanding the zero-or-more operation around (, EXPR)$*$ by creating two new rules $\text{p}_1 \rightarrow \epsilon$ and $\text{p}_1 \rightarrow \text{p}_1$(, EXPR). The rule write is then changed to write $\rightarrow$ EXPR p_1.

After these two rules have been expanded we are left with the grammar shown in figure 5.4 which is in BNF form.

5.2 Sentence Construction

All the algorithms described in this dissertation produce sentential forms that must transformed into sentences from the grammar for which the sentential forms were constructed. Here we described how to construct, for each nonterminal symbol in a grammar, a string of terminals that can be derived from that nonterminal symbol and be used to construct a sentence in a grammar.

The algorithm to perform the construction of terminal sentences for each nonterminal is shown in algorithm 5.1. It works by iterating over each production in the grammar until it finds a production, p, where the body of the production contains only terminal symbols (line 7). Once this production has been found it is marked as complete and the algorithms iterates through all the productions, q, where the nonterminal at the head of p is in the body of q. p is then replaced by it's body, α, in q. This continues until all the nonterminals have been marked as complete at which point the algorithms breaks out of the loop and returns the terminal strings for each nonterminal.

As an example let us consider the example grammar used in chapter 2. The algorithm will construct the set of terminal string for grammar as follows:

Iteration 1 The algorithm iterates over all the production rules in the grammar. The rule $F \rightarrow \textbf{id}$ is a rule that consists of only terminals so complete$[F]$ = **True** and the rules $T \rightarrow F$ and $T \rightarrow T * F$ are changed to $T \rightarrow \textbf{id}$ and $T \rightarrow T * \textbf{id}$.

Iteration 2 The algorithm iterates over all the production rules in the grammar. The rule $T \rightarrow \textbf{id}$ is identified as a rule that consists of only terminals symbols. complete$[T]$ = **True** and the rules $T \rightarrow T * \textbf{id}$, $E \rightarrow T$ and $E \rightarrow E + T$ are changed to $T \rightarrow \textbf{id} * \textbf{id}$, $E \rightarrow \textbf{id}$ and $E \rightarrow E + \textbf{id}$

Stellenbosch University https://scholar.sun.ac.za

Iteration 3 The algorithm iterates over all the production rules in the grammar. The rules $E \rightarrow \textbf{id}$ is identified as a rule that consists of only terminal symbols so complete$[E] = \textbf{True}$. Then the rules $S \rightarrow E$, $E \rightarrow E + \textbf{id}$ and $F \rightarrow (\ E\ )$, are changed to $S \rightarrow \textbf{id}$, $E \rightarrow \textbf{id} + \textbf{id}$ and $F \rightarrow (\ \textbf{id}\ )$.

After the third iteration is complete all the rules in the grammar have been assigned so that we have.

$$S \rightarrow \textbf{id}$$
$$E \rightarrow \textbf{id}$$
$$E \rightarrow \textbf{id} + \textbf{id}$$
$$T \rightarrow \textbf{id}$$
$$T \rightarrow \textbf{id} * \textbf{id}$$
$$F \rightarrow \textbf{id}$$
$$F \rightarrow (\ \textbf{id}\ )$$

The algorithm then returns the shortest sentence available for each nonterminal symbol in the gammar. In this example every nonterminal can be rewritten as the terminal **id**.

Algorithm 5.1 Construction of a terminal string for every nonterminal

```
1:  procedure TERMINAL STRINGS(G)
2:      complete = { n: False for n in N_G}
3:      sentences = P_G
4:      while True do
5:          change = False
6:          for p → α in sentences do
7:              if t ∈ T_G for all t ∈ α then
8:                  complete[t] = True
9:                  for q → βpγ in sentences do
10:                     q = βαγ
11:                 end for
12:             end if
13:         end for
14:         if complete[n] is True for all n in N_G then
15:             break
16:         end if
17:     end while
18:     return shortest terminal string for each production
19: end procedure
```

5.3 LR(1) Parse Table Generation

The parsing automata and their associated tables were generated using the methods described in [2]. The description of the closure, *closure*(I), for a set

Stellenbosch University https://scholar.sun.ac.za

of items, I, is given in definition 2.3.2. The description of the *goto* function is given in definition 4.7. These two definitions are given in terms of the construction of an LR(0) automaton, but LR(1) parsing automata were searched to generate sentences that coverage criteria. The construction of the LR(1) parsing automata, which includes the computation of the closure and goto sets, is given in the following section.

While for LR(0) parsers the items were defined as $A \rightarrow \alpha \cdot \beta$, for LR(1) we must incorporate the lookahead symbol into the definition of an item. An LR(1) item is defined as $[A \rightarrow \alpha \cdot \beta, x]$, where x is the lookahead symbol for the item. The terminal symbol, x is always in $\mathit{follow}(A)$.

When $\beta \neq \epsilon$ in an item $[A \rightarrow \alpha \cdot \beta, x]$ the parser performs a shift or goto operation and the lookahead has no effect. When $\beta = \epsilon$, however, the parser must perform a reduction by the rule $A \rightarrow \alpha$ only when the next input symbol is x. This means the LR(1) parser can perform reductions by different rules depending on what the next input symbol is. An LR(0) parser on the other hand does not consider the next input symbol when performing reductions.

LR(1) Closure

The LR(1) computation of the closure is given in algorithm 5.2.

Algorithm 5.2 Algorithm to compute the LR(1) closure for an item set I

```
1:  procedure LR(1) CLOSURE(I: Item Set)
2:      change = True
3:      while change = True do
4:          change = False
5:          for [A → α · Bβ, x] ∈ I do
6:              for B → γ ∈ P_G do
7:                  for b ∈ first(βx) do
8:                      I = I ∪ [B → γ, b]
9:                      change = True
10:                 end for
11:             end for
12:         end for
13:     end while
14:     return I
15: end procedure
```

LR(1) Goto

The LR(1) computation of the *goto* function is given in algorithm 5.3.

Parsing Automata Construction

The construction of the parsing automata for a LR(1) parser is done by computing GOTO(I, X) for an item set for all the tokens in the grammar to construct all the item sets that may be transitioned to from the original item set for which the closure was computed. We can then use the item sets that were constructed

Stellenbosch University https://scholar.sun.ac.za

Algorithm 5.3 Computation of the LR(1) *goto* function

```
1: procedure LR(1) GOTO(I: Item Set, B ∈ T_G ∪ N_G: Token)
2:     J = ∅
3:     for [A → αB · β, x] do
4:         J = J ∪ [A → ·Bβ, x]
5:     end for
6:     return LR(1) CLOSURE(J)
7: end procedure
```

by the goto function to compute the item sets and transitions that are associated with the new item sets. This process is continued until no new item sets can be constructed, at which time the construction of the LR(1) parsing automata is complete. The item set CLOSURE($\{[S' \to \cdot S, \$]\}$) is the starting item set for all parsing automata so the process is started with this item set.

Algorithm 5.4 shows how to construct an LR(1) parsing automata. In this algorithm PT is a dictionary that maps an item set to a dictionary. The dictionary being mapped to by an item set maps a token in the grammar to a number. This number represents the action that must be taken by the parser. If the number is more or equal to zero, the parser will shift to the state identified by the number. If it is less that zero the parser will reduce by the rule identified by the absolute value of the number.

Algorithm 5.4 Computation of the LR(1) parsing automata

```
1: procedure PARSING AUTOMATA(G: Grammar)
2:     state = 0
3:     C = {state : CLOSURE({[S' → ·S, $]})}
4:     PT = {state : dict()}
5:     change = True
6:     while change is True do
7:         change = False
8:         for item set I ∈ C do
9:             for x ∈ T_G ∪ N_G do
10:                goto = GOTO(I, x)
11:                if goto ≠ ∅ then
12:                    if goto ∉ C then
13:                        state = state + 1
14:                        C[state] = goto
15:                    end if
16:                    i = t where C[t] = goto
17:                    PT[i][x] = goto
18:                    change = True
19:                end if
20:            end for
21:        end for
22:    end while
23: end procedure
```

Stellenbosch University https://scholar.sun.ac.za

5.4 Output

Test cases for a grammar are output in two formats. Each test case can be output as an individual file that may be used as an input file to a software system that accepts the grammar. The test cases may also be collected, per algorithm, in JSON files that may be used for batch processing of all the test cases. This section will provide a precise specification of both output options.

Text When the test cases are output as text each file output by the framework contains a single valid sentence in the language of the target grammar. The test cases are organised by algorithm. If test cases using Purdom's algorithm and LR-automata search are generated two directories, `purdom` and `lrpos`, will be created in the directory of input grammar and each test case that is constructed will be placed in the directory corresponding to the method that was used to generate it. The generated files are numbered by the order in which they were output and are given the extension `.out`. The first test case to be output will be named `0.out`, the second `1.out` and so on.

JSON When the test cases are output as JSON files they are collected by the method that was used to generate them and a JSON object is created to hold them. No directory is created in the directory of the input grammar, instead a file is created for each method of generated that was used. If the WPLR and PLL coverage criteria are coverage by the test cases, two JSON files will be created and name `wplr.json` and `pll.json`.

This constructed JSON is structured as follows.

```
{
  pos: [True | False],
  alg: <algorithm name>,
  test_cases: [
    <test case 0>, <test case 1>,..., <test case n>
  ]
}
```

Stellenbosch University https://scholar.sun.ac.za

Chapter 6

Evaluation and Interpretation

The software system and algorthms implemented for this dissertation were tested in two scenarios. Firstly, the software was applied in a classroom setting to faciliate the automatic test case generation and marking of student submissions. Secondly the algorithms were evaluated against one another to gather and compare empirical performance data.

Each test input set will be referred to by the name of the coverage criteria that it satisfies, except the set of test cases generated by Purdom's algorithm. Even though the coverage of the PLR and NLR coverage critera is not guaranteed positive and negative test inputs generated through LR-automata search will be refered to PLR and NLR test input sets.

6.1 Classroom Evaluation

Here testing was performed using a number of parsers that were implemented by students in two university level compiler courses, one taught at Southampton in 2010 and the other at Stellenbosch in 2015. In both courses the students were required to implement a parser using a parser generator. Yacc and ANTLR were the prescribed generators.

The purpose of this testing was to explore whether or not a set of automatically generated test cases would yield results that more accurately determined the quality of the parsers implemented by the students. This is motivated by a number of advantages of automatically generated test cases when compared to handwritten test cases, which are briefly discussed in section 1.1.3. The quality of a set of generated test cases may be superior to hand written test cases for the following reasons.

Human error The automatic generation of test cases completely removes the possibility of intro ducting human errors. We can be certain that the positive and negative test cases constructed by the software are included and excluded from the language generated by the grammar.

Quantity The number of test cases that can be generated is much larger than can be expected to be handwritten. Depending on the size of the grammar

Stellenbosch University https://scholar.sun.ac.za

the test cases may number in the thousands which is not feasible for a human to reproduce in a reasonable amount of time. The satisfaction of the coverage metrics also ensure that each test case tests a specific aspect of the grammar, the PLL coverage criteria for example specifically test the implementation of each rule in the grammar. The generated test cases are more granular, resulting a larger variation in the pass rates that may be achieved by each student.

Coverage The algorithms that are implemented in this dissertation ensure that the sets of test cases we generate satisfy the coverage metrics discussed throughout and as such we can be confident that, when taken together, all the test cases are representative of the language generated by the given grammar.

As shown shown in table 6.1, orders of magnitude more test cases can be generated than can be feasibly constructed by hand. The generated sets of test cases and the coverage metrics they satisfy also constitute a much more comprehensive set of test cases than the handwritten test cases, which may not be representative of the language generated by the CFG being parsed.

While YACC is an LALR parser generator, the method of LR-automata search is applied to an LR(1) automata generated for the grammar. This larger automata results in a larger test set.

6.1.1 Methodology

For both languages we used reference grammars to generate the test cases. For the Simpl language we used the reference grammar (see Appendix A). For the Niklaus language a reference grammar was contracted from the textual description of the grammar (see Appendix B). For each of the available grammars and submissions the same test procedure was followed with the purpose of computing the positive and negative pass rates for each student with respect to each set of test cases, handwritten and generated. The pass rate for a submission with respect to a set of test cases is the percentage of test cases that were correctly classified as positive or negative by the submission.

For each of the grammars six sets of test cases were generated that satisfied each of the coverage metrics discussed in this dissertation. The number of test cases generated for each test case set is shown in table 6.1. For the group student submissions that parsed the Niklaus we investigate the distribution of pass rates and for the group of student submissions that parsed the Simpl grammar, due to the small size of the group, we investigate pass rates on a per student basis.

When reference is made to the group of submissions implementing the Niklaus grammar it will be referred to as the Niklaus group and the group of submissions implementing the Simpl grammar will be referred to as the Simpl group.

Niklaus (63 Submissions)

After the collection of the past rates for each of the coverage criteria per student we constructed histograms to investigate their distribution.

Sturges' rule [24] was used to approximate the number of bins that would be needed. The rule states that $K = 1 + \log_2(N)$, where K is the number of bins and N is the number of samples. Using this rule we have $1 + \log_2(63) = 6.97 \approx 7$

Stellenbosch University https://scholar.sun.ac.za

# Positive Test Cases		
Algorithm	Simple	Niklaus
Purdom	2	23
PLL	256	110
WPLR	1037	532
PLR	1163	554
Handwritten	89	25
# Negative Test Cases		
NLL	6500	6539
NLR	13621	7227
Handwritten	44	25

Table 6.1: Number of test cases generated using each algorithm for each CFG

bins for the distribution of Niklaus pass rates We decided to use 10 bins for Niklaus data as this number is divides the domain of pass rates, $[0, 100]$, with no remainder. As our data is very skewed we also provide histograms for each coverage criteria showing the distributions of pass rates in the range $[90, 100]$. Three bins were selected for this range.

To investigate the correlation between those pass rates obtained from a handwritten set of test cases and those obtained from a generated set of test cases we constructed scatter plots showing the pass rates of each generated set of positive test cases against the handwritten set of test cases and each generated set of negative test cases against the handwritten negative test cases. We also calculated the Pearson's correlation coefficient for the data sets used in each of the scatter plots.

Simpl (11 Submissions)

After the collection of the pass rates based on each of the coverage criteria per student we constructed bar graphs to investigate how well each student performed in each of the coverage metrics. The bar graphs, exhibited in figures 6.14 – 6.16 show, per student, the pass rate for each coverage criteria.

The order of the bars in each of the bar graphs that is constructed with respect to a generated set of test is the same as the order for bar graphs constructed from the positive or negative handwritten set of test cases. This allows us to investigate whether the performance of each submission changes based on whether they are evaluated with a handwritten or generated set of test cases.

6.1.2 Case Study: Niklaus

In this case study students were required to implement, from a textual description of the grammar, for the Niklaus language, given in Appendix B.1. This a small imperative programming language. It contains a number of well known constructs, such as conditional statements (if-else), loops (while) and input-output (read/write) expressions. The language is also equipped with functions and procedure, where functions return a value and procedures do not.

This assignment was given in the 2010 course at the University of Southampton to a group of undergraduate students in their second academic year of study.

Stellenbosch University https://scholar.sun.ac.za

Handwritten test cases were used for evaluation, with 25 positive and 25 negative test cases being provided by the examiner.

Handwritten Test Cases

The distribution of pass rates based on the handwritten positive test cases is shown in Figure 6.1a and the distribution of pass rates based on the negative test cases is shown in Figure 6.1b. When these two distribution are inspected it is clear that higher pass rates are more likely with both distributions being skewed to the right. The distribution for negative handwritten test cases, however, is skewed much more so, with the lowest achieved pass rate being 75% and 58 of the 63 students achieving pass rates of 80% or higher.

We also see the distribution of pass rates based on the positive handwritten test cases in the range $[90, 100]$. With 25 test cases the only pass rates achievable in this range are 92%, 96% and 100%, these pass rates are represented by the 3 bins in the histogram. For the negative test cases we see the distribution of pass rates based on the negative test cases in the range $[90, 100]$ on the right side of Figure 6.1b. Once again, three bins were used here as there are 25 test cases. Here we see that while most students achieved a high pass rate, the majority of them did not achieve a pass rate of 100%, with 13 submissions achieving this.

The distributions for the hand written test cases indicate that the pass rate for submissions is skewed to the right, with the most submissions in each set of test cases achieving pass rates above 90%.

We believe that because the students were required to implement the grammar using a parser generator and not from scratch higher pass rates are more likely. The students do not have to grapple with the complexities of a low-level implementation, but can focus on the rules and tokens in terms of their high-level structure.

Purdom's Algorithm

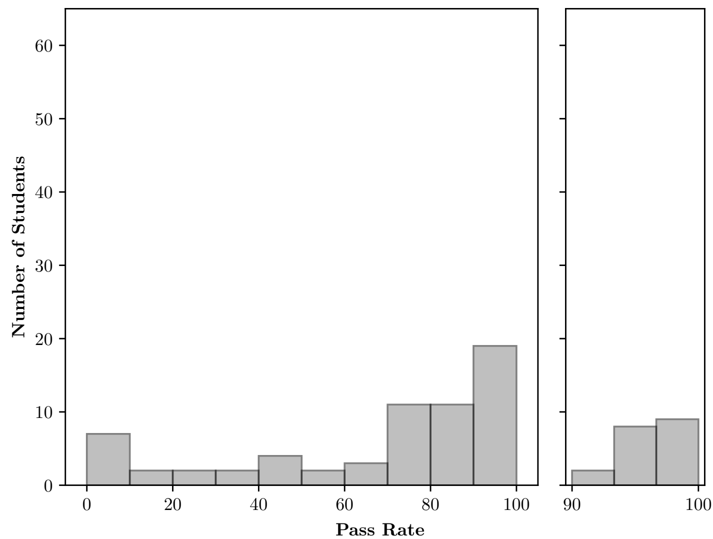
Figure 6.2 shows the distribution of pass rates based on the test cases generated using Purdom's algorithm. As with the handwritten positive test cases, the distribution peaks in the range $[90, 100]$. The peak is however higher with more than 40 submissions in the range, 22 which achieved a pass rate of 100%.

There are more submissions that achieved a pass rate of less 10% with 4 submissions not passing any of the test cases in the set. This may be due to the fact that Purdom's algorithm will produce test cases that simultaneously test as many productions as possible. This increase in test complexity explains the increase in the number of submissions that performed poorly.

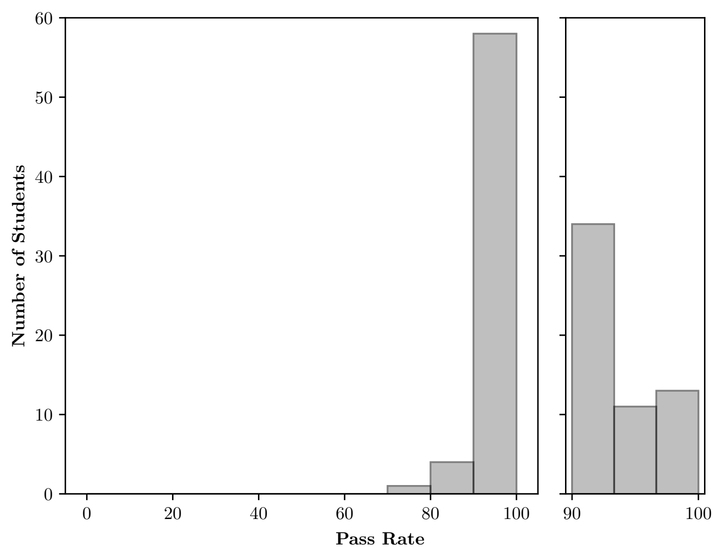
When inspecting the distribution of the pass rates in the $[90, 100]$ we see a similar trend to that of the handwritten positive test cases in the range $[90, 100]$, with more students falling into this range.

Figure 6.3 shows the correlation of the pass rates achieved for positive test cases generated using Purdom's algorithm and those that were handwritten with the line of linear regression. We see that there is some correlation. The large majority of submissions that achieved a pass rate of more than 50% for the handwritten test sets achieving a pass rate of more than 80% for the test set generated using Purdom's algorithm. The Pearson correlation coefficient in Figure 6.3 is 0.89. The equation for the linear regression line, $y = 14.72 +$

Stellenbosch University https://scholar.sun.ac.za



(a) Distribution of the pass rates attained for the handwritten positive test cases.



(b) Distribution of the pass rates attained for the handwritten negative test cases.

Figure 6.1: Niklaus (N=63): Distribution of pass rates for the handwritten test cases

Stellenbosch University https://scholar.sun.ac.za

$0.95x$, indicated a constant increase in achieved pass rates when using Purdom's algorithm.

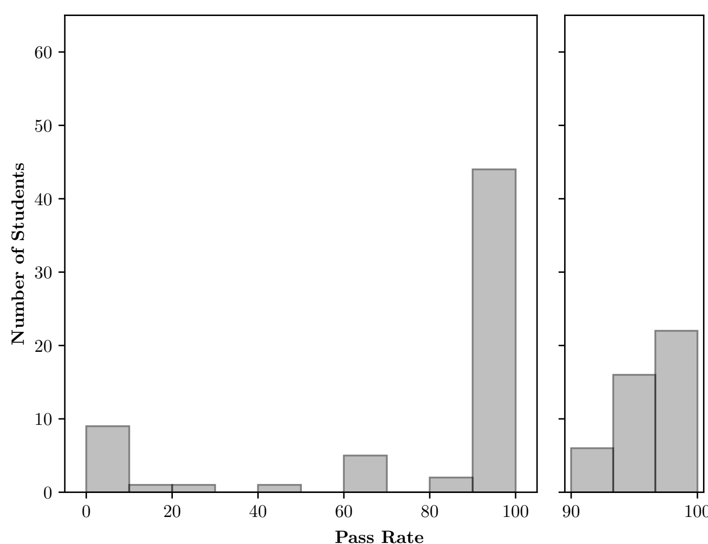


Figure 6.2: Niklaus (N=63): Distribution of the pass rates attained for a test set generated using Purdom's algorithm

PLL

Figure 6.4 shows the distribution of pass rates based on the test set that satisfies the PLL coverage criteria. Here we see that the number of submissions that achieve a pass rate of less than 10% and more 90% is again increased in comparison to the distribution of the handwritten test cases. 10 submissions achieved pass rates of 0% and 27 achieved pass rates of 100%.

When we look at the distribution of pass rates in the range $[90, 100]$ we see the same trend as in the above distributions where more submissions achieve pass rates of 100%.

Figure 6.5 shows the correlation of the pass rates achieved for the positive test cases generated that satisfy the PLL coverage criteria and those that were handwritten. As with the test cases generated using Purdom's algorithm, we see that there is some correlation. The Pearson correlation for these two sets is 0.85 which is slightly less than that for Figure 6.3. The equation for the linear regress line in $y = -4.68 + 1.07x$. This is closer than Purdom's algorithm to the ideal line of $y = x$.

WPLR

Figure 6.6 shows the distribution of pass rates based on the test set that satisfies the WPLR coverage criteria. We see a similar distribution to that of Figure 6.4

Stellenbosch University https://scholar.sun.ac.za

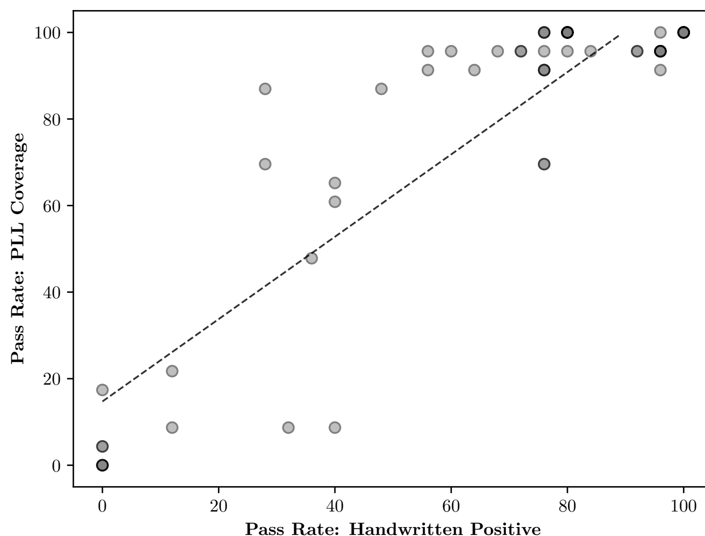


Figure 6.3: Scatter plot showing the correlation between handwritten pass rates and those attained for a test set generated using Purdom's algorithm. The darker dots indicate that more students achieved the same pass rate for both test sets.

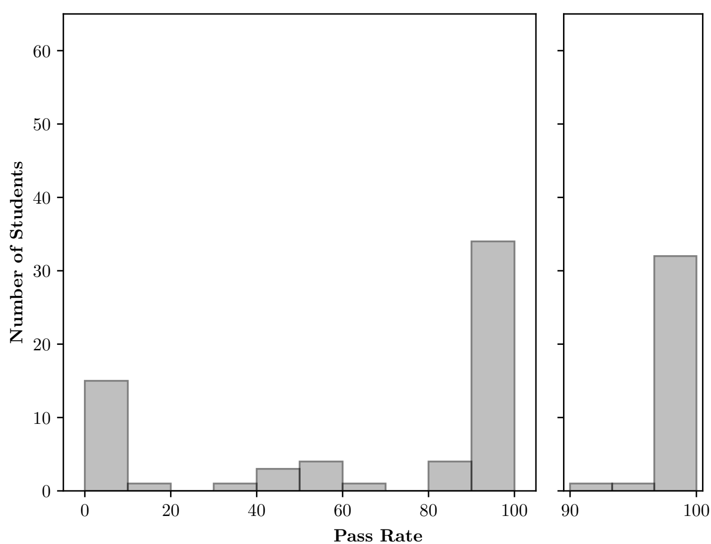


Figure 6.4: Niklaus (N=63): Distribution of the pass rates attained for a test set that satisfies the PLL coverage criteria.

Stellenbosch University https://scholar.sun.ac.za

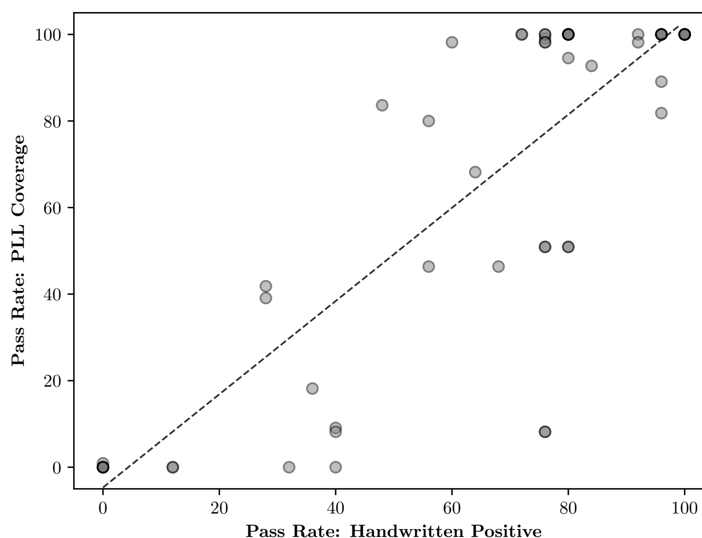


Figure 6.5: Scatter plot showing the correlation between handwritten pass rates and those attained for a test set generated that satisfies PLL coverage.

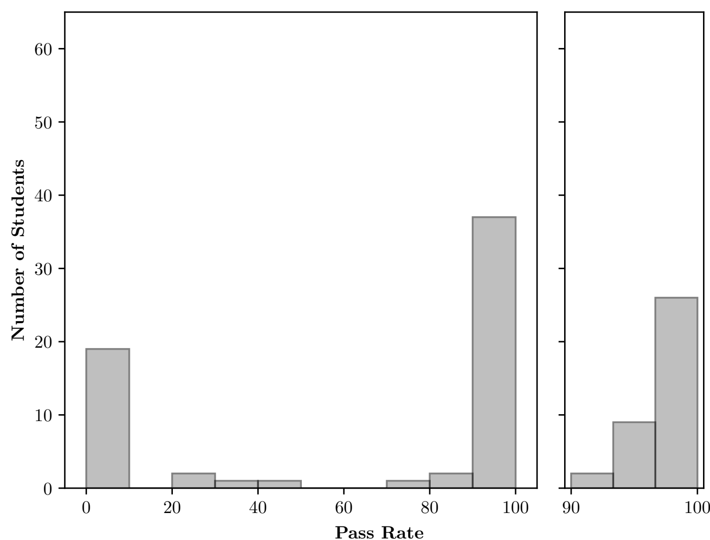


Figure 6.6: Niklaus (N=63): Distribution of the pass rates attained for a test set that satisfies the WPLR coverage criteria.

with most submissions achieving pass rates of less than 10% or higher that 90%. Here we had 18 students that achieved pass rates of 100% and 10 that achieved

Stellenbosch University https://scholar.sun.ac.za

Figure 6.7: Scatter plot showing the correlation between handwritten pass rates and those attained for a test set generated that satisfies WPLR coverage.

pass rates of 0%.

For the range $[90, 100]$, we see that the pass rates here are more evenly spread than those for Purdom and PLL coverage towards the maximum pass rate of 100%, where most submissions fell.

Figure 6.7 shows the correlation of pass rates achieved for the positive test cases generated to satisfy the WPLR coverage criteria. These two sets show the smallest correlation, with a Pearson correlation coefficient of 0.77. The regression line for this scatter plot is $y = -6.14 + 1.05x$, which is close to the ideal line, but it has the highest standard error of 0.112, compared to the standard of Purdom's algorithm and PLL coverage at 0.066 and 0.085 respectively. Here we also that for the submissions that achieved pass rates of 10% or less for the test set there is very little to the pass rate achieved for hand written test cases.

Even though this test set satisfies a coverage criteria that is not applicable to the LL(*) parsers that are generated by ANTLR, the results achieved by the students are applicable to the case study. The increased complexity and variation in the sentential forms that are produced by the WPLR, PLR and NLR criteria when compared to the production, PLL and NLL criteria may result in the LR methods of test input generation being better suited as a replacement for handwritten test inputs regardless off what type of parser generator is used to implement the grammar.

PLR

Figure 6.8 shows the distribution of pass rates based on the PLR test input set. We see in this distribution that once again the most submissions achieved pass rates of 90% or more. With 20 submissions achieving a pass rate of 100% and

Stellenbosch University https://scholar.sun.ac.za

11 submissions achieving pass rates of 0%. For the distribution of pass rates in the range $[90, 100]$. As with the other distributions, most of the submissions achieved pass rates in the highest bracket of 99%–100%.

Figure 6.9 show the correlation of the pass rates achieved for the PLR test input set and the handwritten test cases. The PLR test inputs show the strongest correlation to the handwritten test cases with a Pearson correlation coefficient of 0.89. The regression line, $y = -3.82 + 1.08x$, is also the closer to the ideal line than all other algorithms in the range [0, 100].

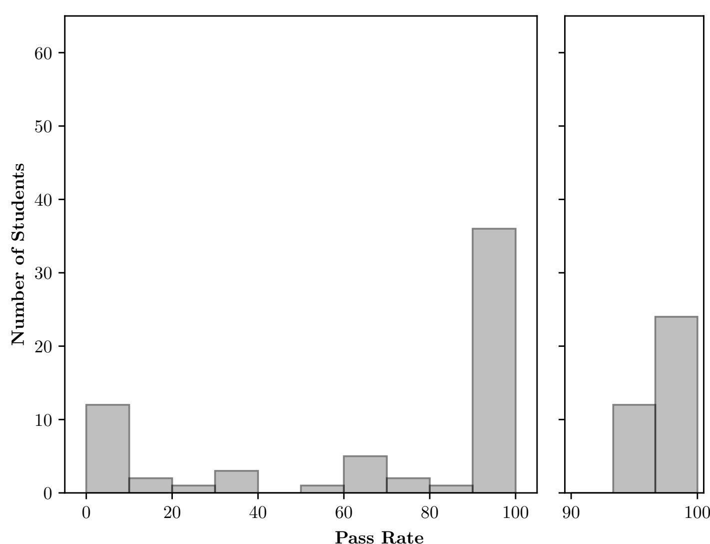


Figure 6.8: Distribution of the pass rates attained for the PLR test input set

NLL

When evaluating the distributions for the pass rates achieved when submissions are tested using the set of test case generated to satisfy the NLR coverage criterion we need to construct only one histogram, shown in 6.12, to see all the relevant data. The figure shows that, as with Figure 6.10, all the submissions except one, submission lgw1e10.1, achieved pass rates of 100%. Submission lgw1e10.1 achieved a pass rate of 93.68%.

Figure 6.11 shows the correlation between the pass rates for the test cases that were generated to satisfy the NLL coverage criterion and those negative test cases that were handwritten. Here we see the lowest correlation, with a Pearson correlation coefficient of 0.58 and a linear regression line of $y = 80 + 0.21x$, which is the furthest from the ideal line of $y = x$.

Stellenbosch University https://scholar.sun.ac.za

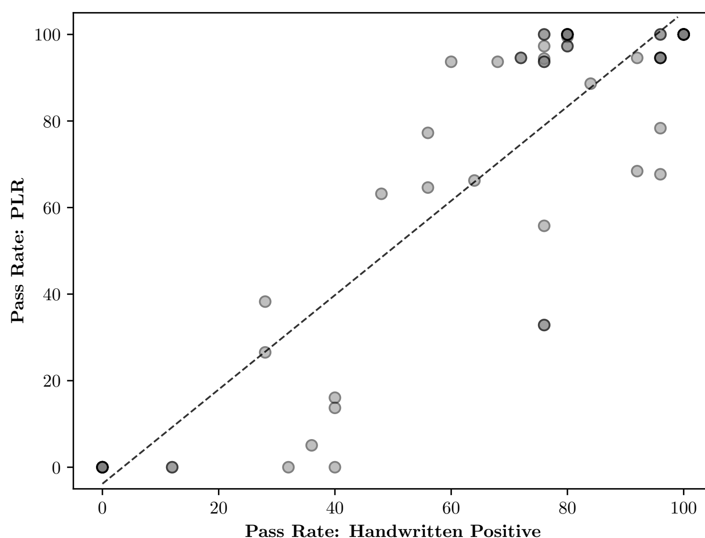


Figure 6.9: scatter plot showing the correlation between handwritten pass rates and those attained for the PLR test input set.

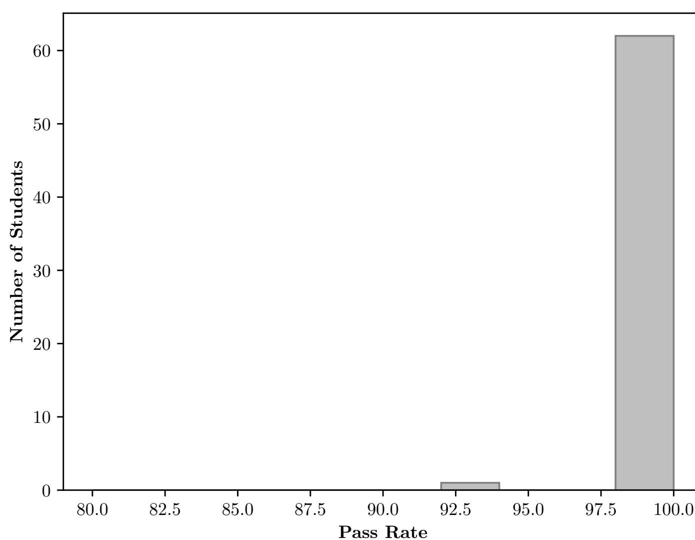


Figure 6.10: Distribution of the pass rates attained for a test set that satisfies the NLL coverage criteria.

NLR

Figure 6.12 shows the distribution of pass rates for negative test cases generated to satisfy the NLR coverage criterion. The x-axis of 6.12, ranging from 80% to

Stellenbosch University https://scholar.sun.ac.za

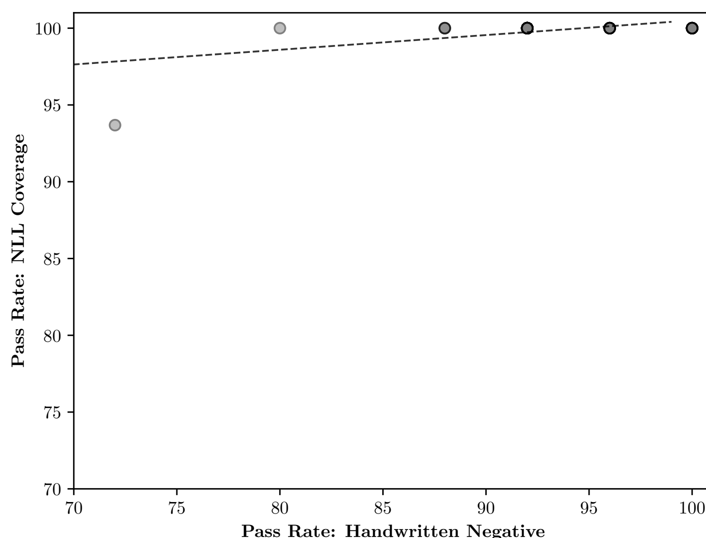


Figure 6.11: Distribution of the pass rates attained for a test set that satisfies the NLL coverage criteria in the range [98, 100].

100%, contains the pass rates for all submissions. We see that data all submissions achieved a pass rate of 98% or more with only one outlier, submission lgw1e10.1, achieving a pass rate of 86.75%.

The histogram to the right shows the distribution of pass rates the negative test cases generated to satisfy the NLR coverage criterion in the range [98, 100]. We see here that, while all submissions achieved pass rates of more that 98%, the majority of submissions achieved pass rates of 100%, with 35 submissions achieving this.

Figure 6.13 shows the correlation between the pass rates for the test cases that were generated to satisfy the NLR coverage criteria and those negative test cases that were handwritten. This scatter plot shows essentially no correlation between the achieved pass rates, with a Pearson correlation coefficient of 0.64 and a linear regression line of $y = 78 + 0.22x$.

6.1.3 Case Study: Simpl

In this case study students were required to implement an LALR(1) parser for a grammar for which they were provided an EBNF description. The LALR(1) description of the grammar is given in appendix A. The assignment was given to a group of postgraduate students in their fourth academic of study as part of a compiler course. The students were required to implement their own handwritten test cases along with the 44 positive and 89 negative test cases constructed by the instructor. Those test cases constructed by the students were not used as part of this case study as they are specific to each submission. For the purpose of evaluation we compare the generated test cases sets to only those test cases constructed by the instructor.

Stellenbosch University https://scholar.sun.ac.za

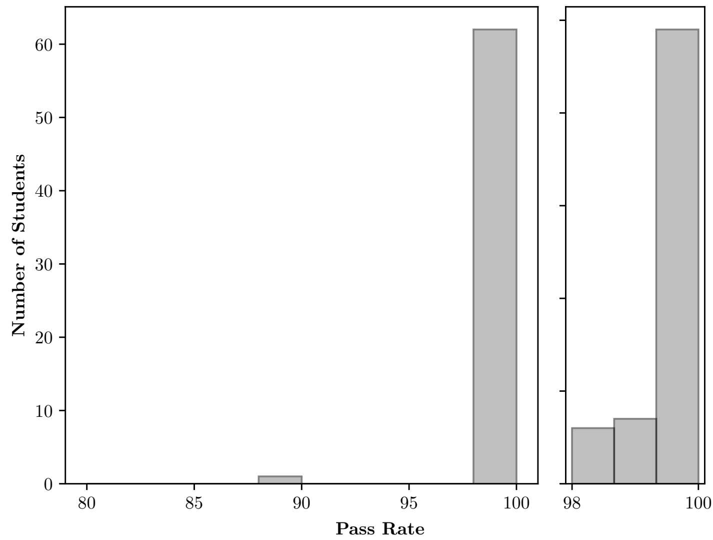


Figure 6.12: Distribution of the pass rates attained for a test set that satisfies the NLR coverage criteria.

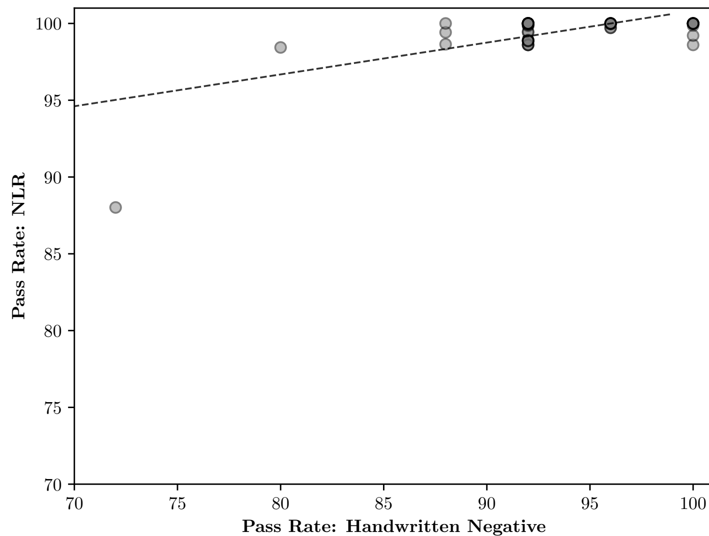


Figure 6.13: Distribution of the pass rates attained for a test set that satisfies the NLR coverage criteria in the range $[98, 100]$.

In the bar graphs the submissions are identified only by the unique number assigned to each submission shown in Appendix **??**. This is done for the purpose

Stellenbosch University https://scholar.sun.ac.za

of readability.

Handwritten Test Cases

The pass rate for each student is shown in Figure 6.14. The bars in each graph are sorted in ascending order by the pass rate attained. We see that the pass rates for the Simpl group over the positive test cases are higher than those of the Niklaus group with an average of 83.35% against an average of 66.41%. The averages for the pass rates over the set of negative handwritten test cases, however, is similar, with the average pass rate for the Simple and Niklaus groups being 92.97% and 93.65% respectively.

The higher average pass rates achieved for the positive handwritten test case cases may be a combination of the task being simpler (as a EBNF description for the grammar is given) and the students being more experienced as they are more academically mature.

Positive Generated Test Cases

The pass rates for each submission when evaluated with each set of generated positive test cases are shown in Figure 6.15.

The pass rates achieved by the submission when evaluated with the test set generated using Purdom's algorithm is shown by the blue bars. Even though this test set consisted of only 2 test cases we see that, for the most part, those that performed well with the handwritten test set performed well here. We also see that no submission failed both test cases as Purdom's algorithm produced one very simple test case using only one production rule and another test cases using all the other production rules. The less complex test case was successfully parsed by all submissions.

The pass rates for each submission when evaluated with a test set that was generated to satisfy the PLL coverage criterion is shown by the orange bars. This more closely resembles the handwritten pass rates and the two methods of evaluation have Pearson correlation coefficient of 0.7. We do, however, see that the pass rates are higher when testing with PLL. This mean pass rate is 90.8% compared to the 83.35% for the handwritten test cases.

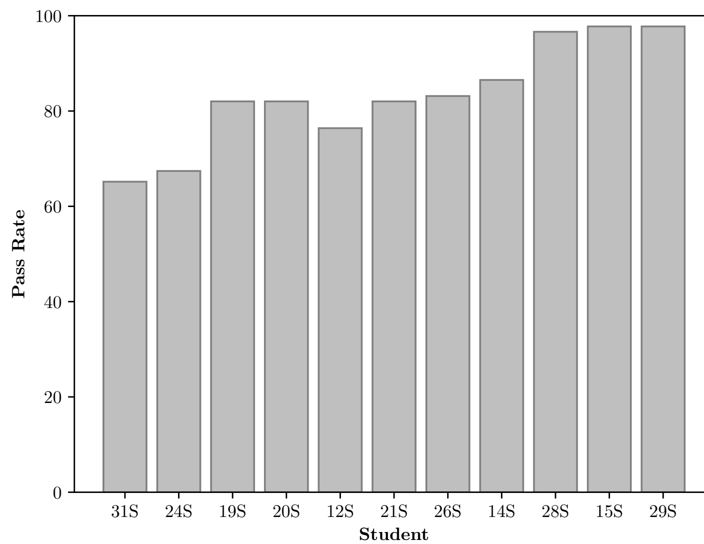
The pass rates for the submissions when evaluated with test cases that were generated to satisfy the WPLR coverage criterion is shown by the green bars. Here we see that the pass rates are once again higher than those for the handwritten positive test cases, with a mean pass rate of 90.9%. The Pearson correlation coefficient for this test set with respect to the handwritten test cases is lower than that of the PLL test cases at 0.52.

The pass rates for the submissions when evaluated with the PLR test input set coverage criterion is shown by the black bars. The pass rates for this set most closely resembles the pass rates for the handwritten test cases with a mean pass rate of 86.4% and the two methods of evaluation have a Pearson correlation coefficient of 0.74.

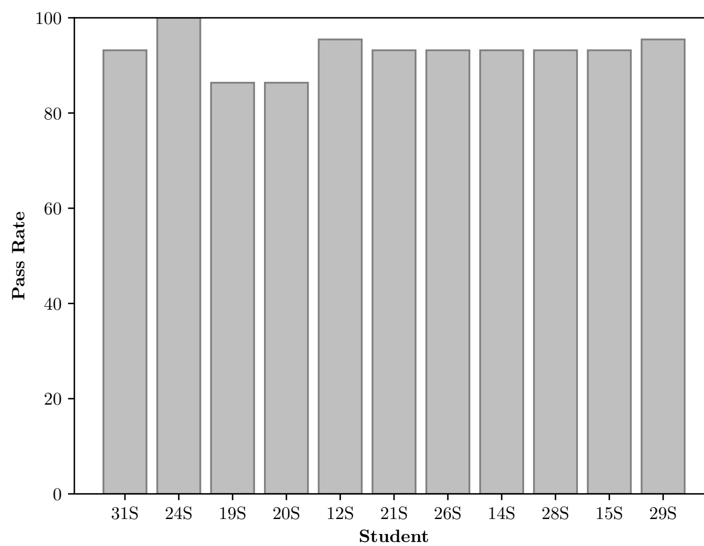
Negative Generated Test Cases

The pass rates for each submission when evaluated with each set of generated generated test cases is in Figure 6.16.

Stellenbosch University https://scholar.sun.ac.za



(a) Handwritten positive test cases



(b) Handwritten negative test cases

Figure 6.14: Bar graphs showing pass rates for each student over the handwritten test cases.

Stellenbosch University https://scholar.sun.ac.za

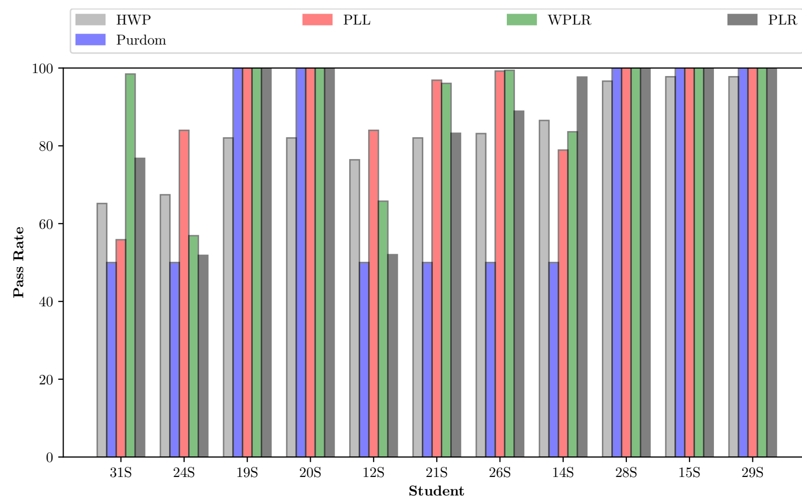


Figure 6.15: Pass rates for each submission evaluated with each set of generated positive test cases

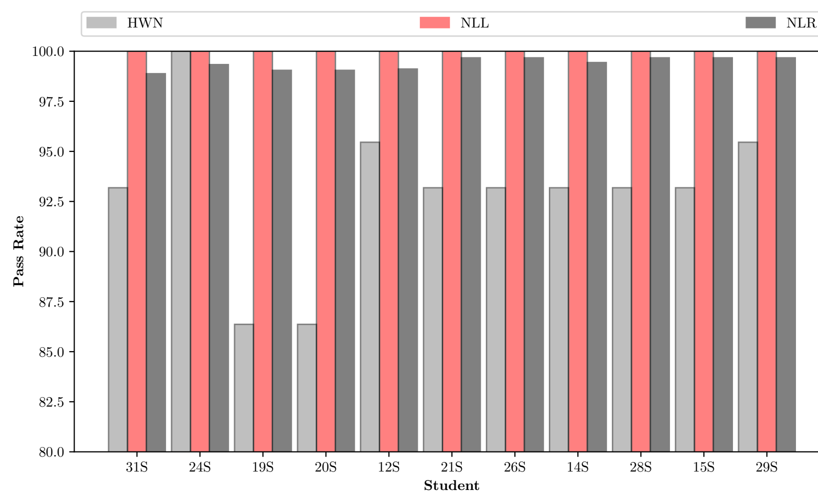


Figure 6.16: Pass rates for each students evaluated with the negative generated test cases

The results of evaluating each student submission with the test sets generated to satisfy the NLL and NLR coverage criteria are indicated by the red and black bars respectively. The pass rates here are similar to those of the Niklaus group. In comparison to the handwritten negative test cases the pass rates are shifted upward dramatically with the NLR test cases having a mean pass rate of 99.38% and the NLL test cases have a mean pass rate of 100%. For both methods of evaluation of evaluation the correlation with the handwritten test cases is also very weak. The NLR test cases have a Pearson correlation coefficient of 0.35 and the NLL test cases have a correlation coefficient of 0.

Stellenbosch University https://scholar.sun.ac.za

6.1.4 Interpretation

After the evaluation of the results of our two case we can conclude that the positive methods of test case generation are much better for assessing the quality of the student submissions than the negative methods. The positive methods show a much stronger correlation to the handwritten test cases than the negative methods, which showed essentially no correlation.

Negative Test Cases

While the two case studies considered in this dissertation were fundamentally different in the manner in which the grammars were implemented. The task for the Simpl grammar being an exercise in transforming an EBNF description of a grammar into an LALR(1) parsable format and the other an exercise in deriving a grammar from a textual description. What is common between the two assignments is that both required the students to implement the grammars using parser generators.

The NLL/NLR coverage criteria construct invalid sentences through the insertion of incorrect tokens. This is not sufficient for the evaluation of parser that was automatically generated as all that is required for such a parser is a description of the grammar in the format of the tool being used. The negative test cases, by virtue of being described at the level of tokens, effectively only test for typos in the grammar that is provided to the parser generator and not for flaws in the rules themselves that would lead to an incorrect description of the grammar. For us to effectively assess the quality of submitted parsers using invalid sentences, the sentences would have to generated through the manipulation of the grammar rules and not the grammar tokens. This would more effectively test the parsers at a higher level and not attempt to expose the types of flaws in the grammar that we have seen are highly unlikely to occur.

A suggested method to mitigate this problem is to weight the passing and failing test cases differently when performing the evaluation or to subtract a constant value e.g., a percentage point, from the total score for each failed test case.

Positive Test Cases

The positive methods of test cases generation, in particular generating positive test case through a search of the parsing automata, yielded much better results than the negative methods of test cases generation. These test cases are much better suited to the task of evaluating submissions as they produce sentences that test all parts of the grammar. The PLL and PLR methods are the most effective.

The method of constructing the test cases to satisfy the PLL coverage does not have the highest correlation to the handwritten test cases in terms of pass rates, but it produces a set of test cases that ensures that all grammar rules are exercised in proportion to the size of the first set of the grammar rule. The test set also exercises one nonterminal at a time in each test case making it possible, if we were to take into account the test cases failed by a submission, to identify where an implementation of a parser may be incorrect.

The PLR test inputs show the highest correlation to the handwritten test cases in terms of pass rates. We consider these test cases produced by this

Stellenbosch University https://scholar.sun.ac.za

algorithm to be of the highest quality as they not only cover all the grammar rules and each token in the first set of every nonterminal in a grammar as with the PLL algorithm, but through the traversal of the parsing automata, the algorithm also generates test cases with most variety in their sentential forms when derived from the start symbol. This can be seen in the example grammar as the PLR method is the only method that will produce a test case with the sentential form $(E+T)$in the example grammar as the PLR method is the only method that will produce a test case with the sentential form $(E+T)$. The test cases combined ensure that every terminal token in the grammar is followed by every terminal in it's follow set in the context of the rule where that terminal occurs. Each test cases also has associated with it a unique path through the parsing automata and the test cases combined represent the smallest number of sentences where each sentence exercises such a unique path.

The PLL test cases may be most effective at exposing flaws in parser implementations and grammar descriptions to the students while the PLR test inputs are most effective when used for the final evaluation of the implementations.

6.2 Performance

Here we tested each implementation of the methods to satisfy the coverage criteria to investigate the performance of each of the algorithms in comparison to one another. For the algorithms applicable to all CFG's, Purdom, PLL, WPLR, NLL, we investigate how differences in the size of a grammar affect the test case generation time. For the PLR and NLR algorithms that are only applicable to LR grammars and are satisfied by traversal of the parsing automata, we investigate how the number of states and transitions affect the test case generation time.

6.2.1 Methodology

To test how the algorithms perform across a range of grammars, a number of grammars were selected from the and the time it took to generate a set of test cases was measured. This was done by taking the average over 100 iterations of each algorithm. The grammars consist of a set of grammars written for ANTLR (taken from the corpus of grammars provided by [1]) and Yacc. The measurements for each algorithm do not include Python start-up time or the time it takes to construct the grammar representation. The measurements were made internally using the available time libraries to only measure how much time it takes to generate a test set. This also does not include the construction of LR parsing automata as the purpose of the testing was to accurately evaluate only the parsing automata search algorithms and not the performance of the automata construction. All measurements were taken to the second decimal place.

For the PLR and NLR methods of test case generation each phase of the algorithms were timed separately to measure their impacts on the total running time.

The size of the grammar is measured as in [15]. There the size of the grammar, $|G|$, is determined to be the, to be the sum of the length of the bodies of all the production rules in the grammar with the head of the rule prepended to

Stellenbosch University https://scholar.sun.ac.za

it.

$$|G| = \sum_{A \in N} \sum_{A \to \alpha} |A\alpha|$$

As the length of the head of a rule is always 1 this may be simplified to

$$|G| = |P| + \sum_{p \to \alpha \in P} |\alpha|$$

To test the PLR and NLR test cases generation methods, two programming languages, C and JavaScript, were used to construct the test grammars. Subsets of the grammars were constructed of different sizes. These grammar subsets were constructed as follows:

1. The start symbol and its rules are taken as the only rules in the subset grammar.

2. For any rule, $A \to \alpha$, in the subset grammar we take one nonterminal, x, from α and add the rules associated with it the grammar.

3. Any nonterminals in the new subset grammar that are not associated with any rules are replaced with equivalent terminal symbols.

The steps above were repeated until a grammar and parsing automata of an appropriate size could be generated, at which point more successor nonterminals and their rules are added to further increase the size of the grammar and parsing automata.

This was done for two reasons. Firstly, to maintain some control over the size of each grammar and the size of the parsing automata generated for it, in particular to ensure that there were enough grammars of varying size to adequately test the PLR and NLR algorithms. Secondly, it was done to avoid the conversion of the ANTLR grammars, which are written to be LL(*), to LR(1). Table 6.2 shows the size data for all the grammars, that is the number of tokens in the grammar, the number of rules in the grammar, the size, $|G|$, of the grammar, and if the grammar was used to test the PLR and NLR test cases generation methods, the number of states and transitions in the parsing automata. Where a subset of grammar, G is used, it is indicated by G_n where n is a number more than or equal to 0. A larger value of n indicates the use of a larger subset of the original grammar.

After the performance results were gathered scatter plots were constructed. For the timings taken for Purdom's algorithm, PLL, WPLR and NLL coverage, the scatter plots investigate the correlation between the time it takes to generate a set of test cases and the size of the grammar for which the test cases are generated. For the PLR and NLR algorithms the scatter plots were constructed to investigate the correlation between the time it takes to generate a set of test cases for a grammar and the size of the parsing automata that was constructed for the grammar, namely the sum of the number of states and transitions.

The performance testing was conducted using a laptop with the reference implementation of Python, CPython 3.6.3, an Intel Core i7 6700HQ with a maximum clock speed of 3.5GHz and 8GB of DDR4 RAM.

Stellenbosch University https://scholar.sun.ac.za

6.2.2 Results

Table 6.3 shows the time it takes, in seconds, to generate sets of test cases that cover the production, PLL, WPLR and NLL coverage criteria for a specific grammar. Tables 6.5 show the time it takes, in seconds, to generate the PLR and NLR test input sets. These tables show the separate measurements for the test cases search, reduction and completion phases as well as the total time taken for the algorithm to complete execution.

Table 6.4 shows the size of each test set that was constructed by each algorithm. All empty entries in the table, except those entries for VBA (Visual Basic for Applications) and VB6 (Visual Basic 6.0), indicate that the algorithm was not tested with the grammar. Testing for VBA and VB6 with the NLL algorithm was not completed due to a lack of RAM.

Grammar (G)	$\|T_G\|$	$\|N_G\|$	$\|P_G\|$	$\|G\|$	Item Sets	Transitions
ANTLR4	53	144	254	664	—	—
ATL	85	233	279	1099	—	—
BASIC	126	198	363	1006	—	—
DCM	28	196	283	783	—	—
Erlang	72	186	332	978	—	—
Java	102	278	539	1472	—	—
Kuka	105	196	344	967	—	—
MDX	51	108	167	427	—	—
Modelica	92	306	489	1301	—	—
Mumps	55	132	246	635	—	—
Pascal	78	167	279	768	—	—
PDP7	113	38	161	348	—	—
VB6	220	754	1557	4128	—	—
VBA	219	705	1484	3910	—	—
WebIDL	82	92	267	722	—	—
C_0	31	14	48	133	112	349
C_1	28	28	83	243	181	481
C_2	38	30	89	258	254	798
C_3	37	34	107	319	330	1259
C_4	60	39	140	408	410	1541
C_5	75	53	177	527	626	2850
C_6	73	55	181	538	769	3903
C_7	71	56	183	544	1129	8232
C_8	79	63	205	621	1205	8840
C_9	80	65	213	677	1360	10172
C_10	80	67	219	703	1495	11447
C_11	82	69	226	725	1560	13997
JavaScript_0	33	23	51	192	433	1951
JavaScript_1	31	27	56	209	598	2898
JavaScript_2	42	37	77	285	896	4911

Table 6.2: The grammars used for testing

Stellenbosch University https://scholar.sun.ac.za

Performance: LL Test Case Generation

Figure 6.17 shows the running time of Purdom's algorithm in comparison to the size of the grammar for which test cases were generated. Purdom's algorithm was the fastest of all the algorithms, completing the test case generation for the largest grammar, VBA, in 0.35s. The correlation between the running time for Purdom's algorithm and the grammar size is strong with a Pearson correlation coefficient of 0.952. The running time of the algorithm is $O(n)$. The gradient of the linear regression line gives us an estimate for a more accurate runtime, which is approximately $6.98 \times 10^{-5}|G|$. The speed of Purdom's algorithm is a result of the fact it does not generate many test cases. The largest set of test cases constructed by this algorithm was for VBA and contained 146 test cases. This is smaller than the smallest test set constructed by any other algorithm.

Figure 6.18 shows the correlation between the running times of the PLL and WPLR algorithms and the size of a particular grammar. These algorithms had similar running times, but performed poorly when compared to Purdom's algorithm, taking approximately a minute to complete the generation of test cases for the VBA grammar. The algorithms did, however produced many more test cases and showed a stronger correlation with the grammar size. The PLL and WPLR algorithms had correlation coefficients of 0.98 and 0.97 respectively. Both had linear running times with the linear regression line indicating a running time of $0.0145|G|$ for the PLL algorithm and $0.176|G|$ for the WPLR algorithm.

The test case sizes constructed by these two algorithms when compared to Purdom's algorithms are much larger. The PLL and WPLR algorithms produced test sets that were orders of magnitude larger than those produced by Purdom's algorithm, with the size of the PLL test sets varying in terms of the number of nonterminals in the grammar and the size of the WPLR test sets varying in terms of the size of the grammar.

Figure 6.19 shows the correlation between the running time of the NLL algorithm and the size of a particular grammar. Here the linear regression does

Grammar	Purdom	PLL	WPLR	NLL
ANTLR4	0.01	0.29	0.33	0.46
ATL	0.03	1.34	1.47	2.24
BASIC	0.02	4	3.77	4.75
DCM	0.01	0.18	0.19	0.29
Erlang	0.06	2.89	2.95	3.32
Java	0.04	4.53	4.2	4.8
Kuka	0.03	1.59	1.61	1.87
MDX	0.01	0.6	0.56	0.74
Modelica	0.04	8.81	8.86	8.85
Mumps	0	0.36	0.38	0.46
Pascal	0.01	0.78	0.84	0.94
PDP7	0	0.04	0.05	0.05
VB6	0.2	47.52	70.14	—
VBA	0.3	52.74	49.41	—
WebIDL	0.01	0.1	0.14	0.11

Table 6.3: Performance of the algorithm over the LL(*) grammars

Stellenbosch University https://scholar.sun.ac.za

not intercept the y-axis at a point close to the origin. Instead, the y-intercept is at -3.31 with the regression line having a gradient of 0.0064. The running of the NLL algorithm also has the lowest correlation to the grammar size with a correlation coefficient of 0.79.

Performance: LR Test Case Generation

Figure 6.20 shows the correlation between the running times, in seconds, of the PLR and NLR algorithms and the size of the parsing automata generated for each grammar, calculated as the sum of the number of states and transitions. The data in the scatter plot show that, empirically, the two algorithms perform as expected. The running times are tightly correlated to size of the automata with correlation coefficients of 0.978 and 0.982 for the PLR and NLR algorithms respectively. They are, however, the slowest algorithms, with the PLR algorithm

Grammar	Purdom	PLL	WPLR	NLL	PLR	NLR
ANTLR4	3	251	396	46169	—	—
ATL	8	936	3118	17320	—	—
BASIC	88	1042	8095	21540	—	—
DCM	3	131	382	7076	—	—
Erlang	13	1797	6927	10493	—	—
Java	3	1323	4201	25903	—	—
Kuka	7	404	1640	18994	—	—
MDX	4	254	632	4358	—	—
Modelica	3	1155	2851	25948	—	—
Mumps	41	376	778	6200	—	—
Pascal	4	407	1059	10821	—	—
PDP7	95	1227	2075	2308	—	—
VB6	119	73775	277215	—	—	—
VBA	146	75767	297336	—	—	—
WebIDL	33	802	2632	6503	—	—
C_0	—	—	—	—	217	949
C_1	—	—	—	—	256	2399
C_2	—	—	—	—	459	3007
C_3	—	—	—	—	687	3856
C_4	—	—	—	—	867	8220
C_5	—	—	—	—	1618	18831
C_6	—	—	—	—	2223	23209
C_7	—	—	—	—	4441	34004
C_8	—	—	—	—	4695	40556
C_9	—	—	—	—	5249	48197
C_10	—	—	—	—	5870	53462
C_11	—	—	—	—	6843	56238
JS_0	—	—	—	—	878	7388
JS_1	—	—	—	—	1167	8836
JS_2	—	—	—	—	2124	16702
JS_3	—	—	—	—	—	—

Table 6.4: Size of each test set constructed for the grammars.

Stellenbosch University https://scholar.sun.ac.za

completing the test set generation for C_11 in an average time of 2.59s and the NLR completing the test set generation in an average time of 5.17s. The PLL and WPLR algorithms, when evaluated with the Pascal grammar, which is comparable in size to C_11, in 0.79s and 0.89s respectively.

Grammars	**PLR**			
	Search	Reduction	Completion	Total
C_0	0.01	0.005	0.005	0.02
C_1	0.014	0.006	0.008	0.028
C_2	0.033	0.012	0.014	0.059
C_3	0.05	0.06	0.026	0.136
C_4	0.07	0.025	0.035	0.13
C_5	0.19	0.05	0.13	0.37
C_6	0.29	0.08	0.3	0.67
C_7	0.84	0.14	0.56	1.54
C_8	0.71	0.18	0.77	1.66
C_9	0.81	0.2	0.86	1.87
C_10	0.75	0.2	0.8	1.75
C_11	0.89	0.23	1.47	2.59
JS_0	0.04	0.03	0.03	0.1
JS_1	0.06	0.04	0.05	0.15
JS_2	0.14	0.08	0.13	0.35
	NLR			
C_0	0.03	0.005	0.005	0.04
C_1	0.07	0.005	0.008	0.083
C_2	0.1	0.04	0.01	0.15
C_3	0.18	0.017	0.022	0.219
C_4	0.31	0.025	0.032	0.367
C_5	0.89	0.06	0.18	1.13
C_6	0.99	0.085	0.314	1.389
C_7	2.01	0.16	0.68	2.85
C_8	2.1	0.21	0.87	3.18
C_9	2.71	0.2	0.89	3.8
C_10	2.27	0.21	0.92	3.4
C_11	3.19	0.26	1.72	5.17
JS_0	0.21	0.03	0.03	0.27
JS_1	0.27	0.04	0.05	0.36
JS_2	0.79	0.08	0.14	1.01

Table 6.5: Performance of PLR and NLR test case generation methods over the LR(1) grammars

Stellenbosch University https://scholar.sun.ac.za

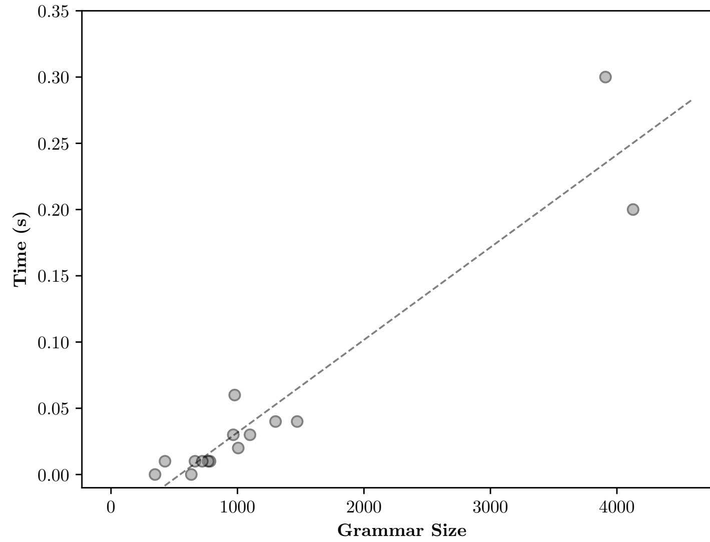


Figure 6.17: Scatter plot showing the correlation between the running time of Purdom's algorithm and the grammar size

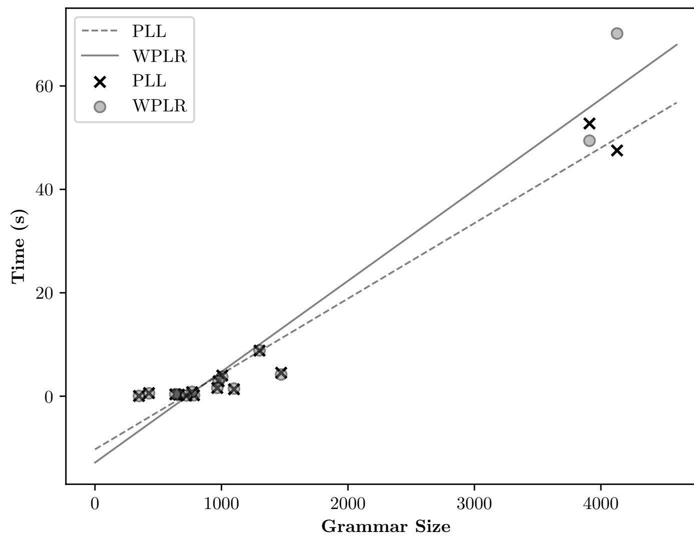


Figure 6.18: Scatter plot showing the correlation between the running time of the PLL and WPLR algorithms and the grammar size

Stellenbosch University https://scholar.sun.ac.za

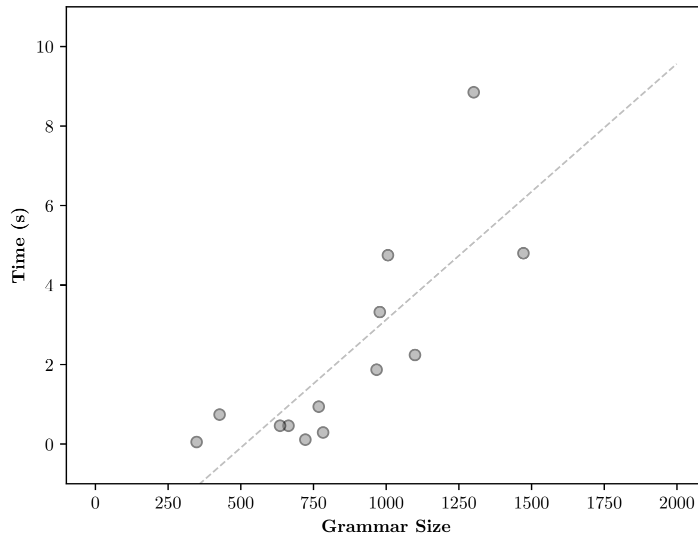


Figure 6.19: Scatter plot showing the correlation between the running time of the NLL algorithm and the grammar size

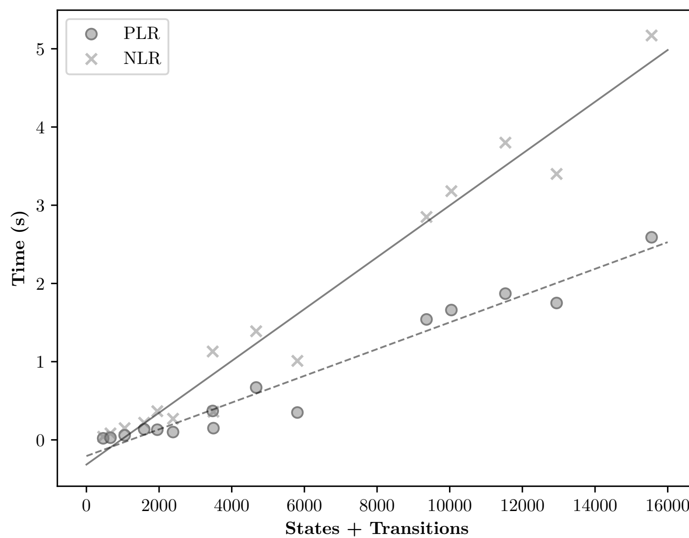


Figure 6.20: Scatter plot showing the correlation between the running time of the PLR and NLR algorithm and size of the parsing automata.

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Chapter 7

Conclusion

In this dissertation we designed and implemented a framework that faciliated the generation of positive and negative test inputs for software systems that act on structured data. The framework made use of test input generation methods that satisfied 4 coverage criteria:

1. Production rule coverage was satisfied with Purdom's algorithm
2. The PLL, WPLR and NLL coverage criteria were satisfied using the methods described by S. V. Zelenov and S. A. Zelenova.
3. The LR-automata algorithms discussed do not guarantee satisfaction of the PLR and NLR coverage criteria. A method to guarantee satisfaction of the coverage criteria could not be obtained, but the sets of positive test inputs generated by the algorithm performed well whe compared to manually constructed test inputs.

The framework and the methods of input generation were evaluated as a means of parser assessment. The generated test inputs were used to assess the quality of 74 parsers implemented by 63 students at Southampton University in their second academic year of study and 11 student at Stellenbosch University in their fourth academic year of study. We found that an assesement based on the test inputs that were generated through LR-automata search coverage criteria correlated most closely to an assement made by test inputs that were constructed by hand.

We also evaluated the performance of all the implements algorithms to determine the factors that influence their running time. We noted that that Purdom's algorithm had the lowest running time and the the running time for it, the PLL, WPLR and NLL test input generation methods varied linearly with respect to the size of the input grammar. The running time of the LR-automata search based methods varied linearly with respect to the size of the constructed LR(1) parsing automata.

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Bibliography

[1] *ANTLR 4: Grammars.* https://github.com/antlr/grammars-v4.git. Accessed: 25-02-2018.

[2] J. D. Ullman A.V. Aho, R. Sethi. *Compilers: Principles, Techniques and Tools.* Addison Wesley Co., 2006.

[3] M Buenen and A Walgude. World quality report 2015–2016. *Capgemini, Sogeti und HP*, 2015.

[4] Eenass Butrus. *Satisfying Coverage Criteria by Grammar Mutations and Purdoms Sentence Generator.* PhD thesis, Masters thesis, Universiteit van Amsterdam, 2014.

[5] Xin Chen and David Pager. Lr (1) parser generator hyacc. In *Proceedings of International Conference on Software Engineering Research and Practice*, pages 471–477, 2008.

[6] F. L. Deremer. Practical translators for lr(k) languages. Technical report, Cambridge, MA, USA, 1969.

[7] Frank DeRemer and Thomas Pennello. Efficient computation of lalr(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649, October 1982.

[8] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. Comparison of context-free grammars based on parsing generated test data. In *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*, volume 6940 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 2012.

[9] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques (Monographs in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] S. C. Johnson. Yacc: Yet another compiler compiler. Technical report, 1975.

[11] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[12] A. J. Korenjak. A practical method for constructing lr (k) processors. *Commun. ACM*, 12(11):613–623, November 1969.

Stellenbosch University https://scholar.sun.ac.za

[13] R. Lämmel. Grammar testing. In *International Conference of Fundamental Approaches to Software Engineering*, pages 201–206. Springer, 2001.

[14] Ralf Lämmel and Wolfram Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In Umit Uyar, Mariusz Fecko, and Ali Duale, editors, *Proceedings of the 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom'06)*, volume 3964 of *LNCS*, pages 19–38. Springer Verlag, 2006.

[15] Martin Lnage and Hand Leibß. To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. *INFORMTICA DIDACTICA*, 8:2008, 2010, 2009.

[16] Brian A Malloy and James F Power. An interpretation of purdom's algorithm for automatic generation of test cases. *International Conference on Computer and Information Science*, 2001.

[17] Brian A Malloy and James F Power. A top-down presentation of purdoms sentence-generation algorithm. *Maynooth, Co. Kildare, Ireland*, 2005.

[18] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[19] David Pager. A practical general method for constructing lr(k) parsers. *Acta Informatica*, 7(3):249–268, Sep 1977.

[20] Asma M Paracha and Frantisek Franek. Testing grammars for top-down parsers. In *Innovations and Advances in Computer Sciences and Engineering*, pages 451–456. Springer, 2010.

[21] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[22] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[23] S. A. Zelenova S. V. Zelenov. Generation of positive and negative test cases for parsers. *Programming and Computer Software*, 31(6):310–320, 2005.

[24] David W. Scott. Sturges' rule. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(3):303–306, 2009.

[25] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

Stellenbosch University https://scholar.sun.ac.za

Appendix A

Simpl

This appendix gives a description of the Simpl grammar for a LR(1) or LALR(1) parser may be constructed.

```
program        →  PROGRAM ID funcdef_l_o body
                  | PROGRAM ID body
funcdef_l_o    →  funcdef funcdef_l_o
                  | funcdef
                  | ϵ
funcdef        →  DEFINE ID '(' param_l ')' funcdef_type body
                  | DEFINE ID '(' param_l ')' body
funcdef_type   →  TO type ID
                  | ϵ
param          →  type array_o ID
                  | type ID
array_o        →  ARRAY
                  | ϵ
param_l        →  param
                  | param ',' param_l
                  | ϵ
body           →  BEGIN vardecls_o statements END
                  | BEGIN statements END
type           →  BOOLEAN | INTEGER
vardecls_o     →  vardecl vardecls_o
                  | vardecl
                  | ϵ
vardecl        →  type array_o id_l ';'
                  | type id_l ';'
id_l           →  ID
                  | ID ',' id_l
statements     →  RELAX
                  | statement_l
statement_l    →  statement
                  | statement_l ';' statement
```

Stellenbosch University https://scholar.sun.ac.za


```
statement        →  ascall
                    | ifstmt
                    | input
                    | LEAVE
                    | output
                    | whileloop
ascall           →  name ascall_rhs_o
                    | name
ascall_rhs_o     →  GETS expr
                    | GETS ARRAY simple
                    | ϵ
ifstmt           →  IF expr THEN statements elsif_l_o else_o END
                    | IF expr THEN statements else_o END
                    | IF expr THEN statements END
                    | IF expr THEN statements elsif_l_o END
                    | ϵ
elsif_l_o        →  ELSIF expr THEN statements elsif_l_o
                    | ELSIF expr THEN statements
                    | ϵ
else_o           →  ELSE statements
                    | ϵ
input            →  READ name
output           →  WRITE output_param_l
output_param_l   →  output_param_t
                    | output_param_t '.'  output_param_l
output_param_t   →  STRING
                    | expr
whileloop        →  WHILE expr DO statements END
expr             →  simple relOp simple
                    | simple
relOp            →  '=' | '#' | '<' | '>' | LTE | GTE
simple           →  negate_o term addTerm_l_o
                    | term addTerm_l_o
                    | negate_o term
                    | term
negate_o         →  '-'
                    | ϵ
addTerm_l_o      →  addOp term addTerm_l_o
                    | addOp term
                    | ϵ
addOp            →  '+' | '-' | OR
```

Stellenbosch University https://scholar.sun.ac.za


```
term           →  factor mulFactor_l_o
                  | factor
mulFactor_l_o  →  mulOp factor mulFactor_l_o
                  | mulOp factor
                  | ϵ
mulOp          →  '*' | '/' | '%' | AND
factor         →  NUMBER | name | '(' expr ')' | NOT factor
                  | TRUE | FALSE
name           →  ID name_access_o
name_access_o  →  '[' simple ']'
                  | '(' factor_l ')'
                  | ϵ
factor_l       →  factor
                  | factor ',' factor_l
```

Stellenbosch University https://scholar.sun.ac.za

Appendix B

Niklaus

B.1 Textual Description

Here we give the textual description of the Niklaus grammar from which students were required to implement a parser using the ANTLR framework. This description deviates from the original description given to the students in the layout used, to make it more readable, and the all parts of the description relating to the later phases of the parsing process, such as the symbol table and other contextual constraints, have been omitted.

B.1.1 Structure

A Niklaus program starts with the keyword MODULE, followed by an identifier and a semicolon (;), a list of declarations that are separated from each other by one semicolon, the keyword BEGIN, a possibly empty, semicolon-separated list of statements, the keyword END, and another semicolon.

B.1.2 Declarations

Declarations are either constant declarations, variable declarations, function declarations, or procedure declarations. A constant declaration consists of the keyword CONST, followed by an identifier, a colon (:), a type, the defined as symbol (:=), and an expression. A variable declaration consists of the keyword VAR, followed by a non-empty, comma-separated list of identifiers, a colon (:), and a type. A function declaration consists of the keyword FUNCTION, followed by an identifier, an opening parenthesis (() the parameter list, a closing parenthesis ()), a colon, a type, the defined as symbol, a possibly empty, semicolon-separated list of variable declarations, the keyword BEGIN, a possibly empty, semicolon-separated list of statements, and finally the keyword END. The elements of the parameter list are separated by a comma (,), and each element of the parameter list consists of a non-empty, comma-separated list of identifiers, a colon, and a type. A procedure declaration is similar to a function declaration, with the difference that it uses the keyword PROCEDURE, and that the colon and type between the closing parenthesis and the defined as symbol are missing. Identifiers are composed of upper- and lower-case letters,

Stellenbosch University https://scholar.sun.ac.za

digits, and the underscore and must start with a letter. All identifiers are case sensitive, i.e., x and X are different.

B.1.3 Types

Niklaus has only three types, truth values (represented by the keyword TRUTH), integers (represented by the keyword COUNT) and double-precision floating-point numbers (represented by the keyword FLOAT).

B.1.4 Statements

Niklaus programs are built from five basic statements (assignments, output statements, conditionals, loops, and procedure calls). A semicolon-separated list of statements can be grouped together using the keywords BEGIN and END, to form a statement by itself:

Assignments An assignment consists of a variable identifier, the defined-as symbol, and an expression.

Output An output statement consists of the keyword OUTPUT, followed by a string literal or by an identifier. OUTPUT writes its argument to the standard output.

Conditionals A conditional consists of the keyword IF, a Boolean expression, the keyword THEN, a statement, the keyword ELSE, and another statement. The ELSE-clause (that is, the keyword and the statement) is optional.

Loops Niklaus contains two kinds of loops, WHILE-loops and REPEAT-loops. A WHILE-loop consists of the keyword WHILE, a Boolean expression, the keyword DO, and a statement. A REPEAT-loop consists of the keyword REPEAT, a statement, the keyword UNTIL and a Boolean expression.

Procedures A procedure call consists of the procedures name, followed by an opening parenthesis, a comma-separated list of expressions, and a closing parenthesis.

B.1.5 Arithmetic Expressions

The basic elements of arithmetic expressions are numbers, constants, and variables:

Numbers Numbers come in two different representations. Integers are just a non-empty sequence of digits, preceded by an optional sign (either "+" or "-"). Double-precision floating point numbers consist of an optional integer, followed by the fractional part (i.e., the decimal point "." followed by an optional sequence of digits), followed by an optional exponent. The exponent starts with the letter "e" (either lower case or upper case), followed by an integer. A zero exponent is allowed for a number. Note that leading zeros are allowed but no space is allowed in a either type of number, not even between the sign and the first digit or between the constituent parts of the exponent.

Stellenbosch University https://scholar.sun.ac.za

Constants Any constant or variable declared to be of type COUNT or FLOAT by itself is already an arithmetic expression.

B.1.6 Boolean Expressions

Boolean expressions are formed from relating two arithmetic expressions with a relational operator. The following relational operators are allowed:

Equality (==),

Non-equality (! =),

Comparisons (<, <=, >=, >).

Any constant or variable declared to be of type TRUTH by itself is already an Boolean expression.

B.1.7 String Literals

A string literal is any sequence of characters (except for a double-quote (″) and a newline) enclosed in double-quotes. String literals can only be used in the OUTPUT statement.

B.1.8 Function Application Expressions

Finally, applying a declared function to a list of arguments also constitutes an expression. A function application consists of the function identifier, followed by an opening parenthesis, a comma-separated list of expressions, and a closing parenthesis. The type of a function application expression is determined by the declared return type of the function.

B.1.9 Comments

Niklaus allows comments to be written by using the symbols /* to begin a comment and */ to end a comment. Comments cannot be nested. Comments and white space can be inserted at any point of the program (except within keywords, identifiers, operators, . . .) without changing the meaning.

Stellenbosch University https://scholar.sun.ac.za

B.2 CFG

Here we give a definition of the grammar that was used to parse the Niklaus grammar described in B.1.

```
program       →  MODULE ID ‘;’
                 decls?
                 BEGIN stmts?  END ‘;’
decls            decl (‘;’ decl)*
stmts            stmt (‘;’ stmt)*
decl          →  const_decl | var_decl | proc_decl | func_decl
const_decl    →  CONST ID ‘:’  type ‘:=’ expr
var_decl      →  VAR ids ‘:’  type
proc_decl     →  PROCEDURE ID ‘(’ params?  ‘)’ ‘:=’
                 (var_decl (‘;’ var_decl)*)?
                 BEGIN stmts END
func_decl     →  FUNCTION ID ‘(’ params?  ‘)’ ‘:’  type ‘:=’
                 (var_decl (‘;’ var_decl)*)?
                 BEGIN stmts END
var_decls     →  var_decl (‘;’ var_decl)*
params        →  param ( ‘,’ param )*
param         →  ids ‘:’  type
ids           →  ID (‘,’ ID)*
type          →  TRUTH | FLOAT | COUNT
stmt          →  if_stmt | assign_stmt | loop_stmt
                 | output_stmt | proc_stmt | BEGIN stmts END
if_stmt       →  IF expr THEN stmt (ELSE stmt)
assign_stmt   →  ID ‘:=’ expr
loop_stmt     →  WHILE expr DO stmt | REPEAT stmt UNTIL expr
output_stmt   →  OUTPUT (STRING_LIT | ID)
proc_stmt     →  ID ‘(’ exprs?  ‘)’
expr          →  arith_expr (RELOP arith_expr)?
arith_expr    →  arith_expr addop term
term          →  term mulop factor
factor        →  ID | num | ‘(’ expr ‘)’
                 | SUB arith_expr | proc_stmt
relop         →  GT | GTE | LT | LTE | NEQ | EQ
addop         →  ADD | SUB
mulop         →  MUL | DIV
num           →  INT | DECIMAL
```

Stellenbosch University https://scholar.sun.ac.za

Appendix C

Results: Nikluas

Student	Positive Algorithms					Negative Algorithms		
	(25) HWP	**(23)** P	**(110)** PLL	**(532)** WPLR	**(554)** PLR	**(25)** HWN	**(6539)** NLL	**(7227)** NLR
kng1g10.2	20	23	104	514	554	2	0	12
gl10g10.2	19	22	109	202	523	2	0	0
kkn1g10.2	25	23	110	532	554	1	0	21
dmjc1g10.1	23	22	110	526	524	2	0	78
kr2g10.2	0	0	0	0	0	0	0	0
idh1g10.3	25	23	110	532	554	2	0	83
ks6g10.1	7	20	43	242	212	2	0	0
sa1g10.3	25	23	110	532	554	2	0	101
temt1g10.3	24	22	110	526	524	0	0	0

Stellenbosch University https://scholar.sun.ac.za

Student	Positive Algorithms **(25)** Handwritten Pos	 **(23)** Purdom	 **(110)** PLL	 **(532)** WPLR	 **(554)** PLR	Negative Algorithms **(25)** Handwritten Neg	 **(6539)** NLL	 **(7227)** NLR
pn3g10.1	20	23	110	532	554	0	0	10
nttp1g10.3	8	2	0	0	0	1	0	0
jz11g10.1	12	20	92	484	350	2	0	42
ijg1g10.2	24	22	110	526	524	2	0	78
as8g10.1	24	22	110	526	524	2	0	0
lgw1e10.1	24	22	110	526	554	7	413	866
ik2g10.1	18	22	110	526	524	2	0	0
md3g10.2	20	22	56	507	539	2	0	0
swb1g10.1	19	23	110	532	554	2	0	5
mk2g10.1	19	23	110	532	554	2	0	101
aed1g10	20	23	110	495	551	2	0	1
ap8g10.1	0	1	0	0	0	1	0	0
je5g09.2	20	23	110	532	554	2	0	0
bdf1g10.1	0	0	0	0	0	0	0	0
dtv1g10.3	24	21	90	5	375	1	0	0
rr15g10.2	0	4	1	125	0	2	0	0
lia1g10.2	0	1	0	0	0	1	0	1
vk2g08.1	19	21	56	5	309	3	0	98
qw3g11.2	19	16	9	20	182	1	0	20
as31g10.4	23	22	108	476	379	3	0	42
gec1g10.1	25	23	110	532	554	2	0	0

Stellenbosch University https://scholar.sun.ac.za

Student	POSITIVE ALGORITHMS					NEGATIVE ALGORITHMS		
	(25) Handwritten Pos	**(23)** Purdom	**(110)** PLL	**(532)** WPLR	**(554)** PLR	**(25)** Handwritten Neg	**(6539)** NLL	**(7227)** NLR
by2g10.2	0	0	0	0	0	0	0	0
ceh2g10.1	19	23	56	511	539	2	0	39
onme1g10.1	25	23	110	532	554	2	0	83
cre1g10.1	18	22	110	526	524	0	0	0
mc11g10.2	25	23	110	532	554	2	0	0
bac2g10.3	24	22	98	508	434	0	0	0
noj1g10.1	24	23	110	532	554	5	0	113
ejh1g10.1	14	22	51	414	428	2	0	9
marvin_niklaus	25	23	110	532	554	0	0	0
jpc1g09.5	25	23	110	532	554	0	0	101
ycc1g11.1	14	21	88	5	358	2	0	4
wb2g10.3	20	23	110	532	554	2	0	0
agps1g10.5	25	23	110	532	554	0	0	0
ts9g10.2	20	23	56	511	539	2	0	101
dg1g10.1	19	16	9	20	182	2	0	0
rjt2g10.1	10	14	10	5	89	1	0	3
yi1g09.1	7	16	46	4	147	1	0	0
tc14g10.1	19	21	108	499	519	2	0	0
ba5g10.1	17	22	51	429	519	2	0	82
jh9g09.1	3	5	0	0	0	2	0	0
pajl1g09.4	20	23	110	532	554	0	0	1

Stellenbosch University https://scholar.sun.ac.za

Student	Positive Algorithms					Negative Algorithms		
	(25) Handwritten Pos	**(23)** Purdom	**(110)** PLL	**(532)** WPLR	**(554)** PLR	**(25)** Handwritten Neg	**(6539)** NLL	**(7227)** NLR
yz7g10.4	3	2	0	0	0	1	0	0
rs8g10.1	19	21	108	499	519	3	0	0
dln1g10.1	10	15	9	20	76	2	0	0
mm1g10.2	24	22	110	526	524	2	0	0
jkv1g10.4	10	2	0	0	0	1	0	0
sgw1g10.1	20	23	110	532	554	1	0	0
cat3g10.1	9	11	20	108	28	2	0	0
sa10g10.1	15	22	108	499	519	2	0	0
rdwc1g10.1	16	21	75	3	367	2	0	81
ap6g10.1	0	0	0	0	0	0	0	0
rw11g09.1	21	22	102	502	491	0	0	57
jb23g10.2	20	23	110	532	554	2	0	0

Stellenbosch University https://scholar.sun.ac.za

Stellenbosch University https://scholar.sun.ac.za

Appendix D

Results: Simpl

Student	Positive Algorithms (89) Handwritten Pos	(2) Purdom	(256) PLL	(1037) WPLR	(1163) PLR	Negative Algorithms (44) Handwritten Neg	(6500) NLL	(13621) NLR
24SimplGrammar	60	1	215	590	603	0	0	91
29Simpl	87	2	256	1037	1163	2	0	45
14simpl	77	1	202	867	1136	3	0	77
19Simpl	73	2	256	1037	1163	6	0	130
28Simpl	86	2	256	1037	1163	3	0	45
20Simpl	73	2	256	1037	1163	6	0	130
15SIMPL	87	2	256	1037	1163	3	0	45
31SIMPL	58	1	143	1021	893	3	0	153
21SIMPL	73	1	248	996	968	3	0	45
12Simple	68	1	215	682	605	2	0	121
26SIMPL	74	1	254	1031	1034	3	0	45