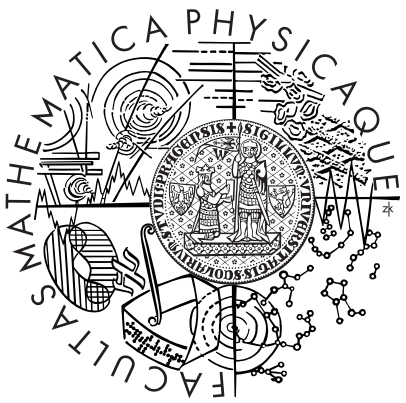
Charles University in Prague
Faculty of Mathematics and Physics

MASTER'S THESIS



Pavol Juhos

Source Code Generator

Department of Software Engineering

Advisor: RNDr. David Bednárek

Study Program: Computer Science

2008

I would like to express my sincere gratitude to all those who helped me during the work on this thesis – specially to my friends Šimon Biľo and Miroslav Kolínsky for their valueable comments and moral support. I would also like to thank my advisor RNDr. David Bednárek for his patience and academic guidance.

I hereby declare that I wrote this thesis on my own, using only the referenced sources. I agree with making my work publicly available.

Prague, August 8, 2008 Pavol Juhos

Contents

Názov práce: Generátor zdrojového kódu
Autor: Pavol Juhos
Katedra: Katedra softwarového inženýrství
Vedúci diplomovej práce: RNDr. David Bednárek
E-mail vedúceho práce: David.Bednarek@mff.cuni.cz

Abstrakt: Pri testovaní prekladačov a iných nástrojov na spracovanie zdrojového kódu zohrávajú obvykle rolu dátových vstupov manuálne vytvárané a zostavované sety skúšobných programov. Príprava takýchto setov v zodpovedajúcej kvalite je však nákladná – ide o náročný a dlhotrvajúci proces. Táto práca skúma možnosti automatizácie procesu tvorby skúšobných programov prostredníctvom generácie zdrojového kódu. Cieľom práce je návrh generátora schopného produkovať korektné programy vo vybranom vysokoúrovňovom programovacom jazyku. V tejto súvislosti práca prezentuje niekoľko teoretických metód a algoritmov na pseudonáhodnú generáciu zdrojového kódu. Metódy zaoberajúce sa bezkontextovými a kontextovo závislými syntaktickými pravidlami sú diskutované samostatne. Praktická využiteľnosť týchto metód je následne demonštrovaná na troch prototypových implementáciách – dvoch generátoroch zdrojového kódu a jednom nástroji na analýzu gramatík. Prvý generátor vytvára reťazce podľa danej stochastickej bezkontextovej gramatiky, druhý umožňuje generovať programy spĺňujúce definíciu jazyka Pascal podľa normy ISO/IEC 7185:1990.

Kľúčové slová: programovacie jazyky, testovanie prekladačov, randomizované testovanie, stochastické bezkontextové gramatiky, generovanie dát

Title: Source Code Generator
Author: Pavol Juhos
Department: Department of Software Engineering
Advisor: RNDr. David Bednárek
Advisor's e-mail address: David.Bednarek@mff.cuni.cz

Abstract: It is a common practice to perform compiler testing with a set of hand-written sample programs. However, gathering larger collections of high-quality test cases is a difficult and time consuming task. This work explores the potential of automatic source code generation techniques to simplify the construction of test suites for compilers and other source code processing tools (e.g. pretty-printers, static code analysis tools, refactoring engines, etc.). The goal is to design a source code generator capable of producing compile-time correct programs in a selected high-level programming language. A set of theoretical methods and algorithms for pseudo-random source code generation is presented. Specific methods are designed for handling context-free and context-sensitive syntax rules of the target languages. The applicability of these methods in practice is demonstrated by the implementation of three prototype tools – two source code generators and one grammar analysis tool. One of the source code generators produces random sentences according to a given stochastic context-free grammar while the other can be used for generation of ISO/IEC 7185:1990 conformant Pascal programs.

Keywords: programming languages, compiler testing, randomized testing, stochastic context-free grammars, test data generation

Chapter 1

Introduction

The field of compiler construction has an important role in the software engineering industry. Since 1950s programs are no longer written in assembly language, but instead they are translated to assembly language from higher-level programming languages by *compilers*. One of the implications of this transition is that the correctness of the final executable code now also depends on the correctness of compilers. Compilers thus became a fundamental part of the software development tool chain.

The conventional ways to achieve high levels of confidence in the correctness of a software system are software testing and software verification. Software verification techniques usually have problems with scaling to larger and more complex software systems. Hence they are rarely considered with regards to compilers. This leaves *software testing* to be the primary method used in this domain.

Software testing can be performed in many forms at different levels [1]. This work focuses on testing compilers as whole programs, i.e., without direct interaction with the internals of their implementation. The technique is also called *black-box testing* [21]. It typically involves presenting the given software system with a collection of test inputs and verifying that the system behaves correctly for each input. In our case the software system under test is a compiler and hence the test data have a form of sample program written in a given programming language.

Conventionally, hand-written sample programs are used for compiler testing. Hence gathering a larger collection of test cases is difficult and time consuming. We suggest that this task could be greatly simplified with source code generation tools capable of producing valid high-level programming language source code without manual intervention.

[1] Unit testing, integration testing, system testing, performance testing etc.

The source code generation approach for automatic construction of compiler test suites has other advantages as well. In our experience automatically generated programs tend to contain unexpected combinations of programming language constructs that were not thought of by compiler authors and in some rare cases not even by the programming language authors. Such programs are very good candidates for exposing intricate defects in compiler implementations. Further advantage of randomized source code generation methods is that they produce different output on each run and therefore provide a literally inexhaustible source of test cases. More on random testing can be found in [41] and [16].

The goal of this thesis is to examine the following three questions:

1. Is it possible to develop such tools at all (in theory and in practice)?
2. What are the possibilities and limitations of source code generation methods?
3. What principles, techniques and algorithms can be used in implementation of source code generators?

In our context the purpose of a source code generator is to produce *strings of symbols* (sequences of characters) that can be interpreted as compile-time correct programs according to the specification of a given programming language. It is important to emphasize that we are primarily concerned with the *form* (syntax) of the produced strings of symbols. The meaning of generated strings as programs is analyzed only to the extent that is required for determining if they satisfy the syntax rules of the target language. In the terminology of programming languages we focus on *syntax* and *static semantics* and largely omit the run-time semantics.

The term "source code generator" can be generally applied to any software tool that produces program source code as it's output. Such definition includes single-purpose code generators that are sometimes used to create programs with a specific function, i.e. programs that perform a specific task. The task is usually specified by the input of the generator. For example parser generators like Bison, ANTLR or JavaCC take a formal (context-free) grammar as an input and produce a parser that recognizes language defined by this grammar. Parser is consider as a program with a specific function.

Despite the apparent appeal to our goal we are not concerned with this kind of tools. Most single-purpose code generators produce structurally similar source code for every input. A collection of test cased created by running such generator on different inputs would therefore exercise only a small part

of the compiler. Hence single-purpose code generators are not appropriate for compiler testing.

This work is also motivated by recent developments in the area of programming languages. In course of the last few years new programming languages have been emerging and gaining popularity more rapidly than before. This trend is also supported by the widespread adoption of virtual machine environments like the Java Virtual Machine (JVM) and the Common Language Runtime (CLR). For illustration recently emerging languages and language implementations include:

- Standalone languages like Ruby, Python, JavaScript, D
- Languages hosted on the JVM – e.g. Scala, Groovy, Jython, JRuby, ...
- Languages hosted on the CLR – e.g. F#, IronRuby, IronPython, ...

Development and integration of new programming languages lead to creation of new compilers and other language processing tools. All of these software systems, and especially compilers, need thorough testing to achieve the reliability required for continuing adoption. To perform such testing, comprehensive compiler test suites are necessary. Since compiler test suite creation is not a simple task, larger test suites are usually not available for emerging languages[2]. We suggest that automatic source code generation tools could be used to create extensive compiler test suites rapidly and thus also help with adoption of new programming languages.

Structure of the text

This text is organized in a following way:

Chapter 2 provides a theoretical background. Here, the concepts of context-free grammars and programming languages are discussed. Basic definitions and notational conventions that are used through-out the text are provided.

Chapter 3 gives a broader definition of the main problem and discusses the scope of this work.

Chapter 4 describes the methods and algorithms for automatic source code generation which constitute the main result of this work.

Chapter 5 describes GRAMGEN, YAGA and PASGEN, the three prototype tools that were created to demonstrate the practical application of the methods and techniques described in the previous chapter.

[2]For established languages like C, C++ and Java comprehensive compiler test suites are usually available or can be purchased from 3rd party vendors.

Chapter 6 provides an overview of existing source code generation methods, implementations of related tools and discusses links with related fields of research.

Chapter 7 reviews the results presented in previous chapters, examines the possible directions for future work and concludes with some final remarks.

Technical details regarding the usage of prototype tools as well as some examples of generated source code are left as appendices.

Chapter 2

Context

2.1 Context-free Grammars

Since our approach to source code generation heavily relies on context-free grammars and their variants, we devote this section to the review of basic definitions and common notation.

Definition 2.1. *Context-free grammar* G is a 4-tuple (V_N, V_T, S, R) where

- V_N is a finite set of non-terminal symbols
- V_T is a finite set of terminal symbols (sometimes called the alphabet and denoted Σ)
- $S \in V_N$ is the start symbol
- $R \subseteq V_N \times (V_N \cup V_T)^*$ is a finite set of production rules[1]

When talking about context-free grammars we will follow the terminology and notational conventions commonly used in the literature [40, 19, 1]. The most important ones are summarized here.

- Production rule $(A, \alpha) \in R$ is usually written as $A \to \alpha$. The non-terminal $A \in V_N$ is called the *head* and the string $\alpha \in (V_T \cup V_N)^*$ is called the *body* of the production rule. We also say that non-terminal A is *associated* with production rule $A \to \alpha$. The body of a production rule can be an empty string which is denoted by ϵ.

[1] The set of production rules is usually denoted as P but to prevent confusion with probability we have chosen to denote it as R.

- When a production rule $A \to \alpha$ is applied to non-terminal A we say that A *rewrites* to α (or equivalently A *expands* to α).
- Sometimes it is convenient to refer to production rules without explicitly stating head and body. In these cases we enumerate production rules as π_i with $i = 1, \ldots, |R|$. In this notation we can write the set of production rules as $R = \{\pi_1, \ldots, \pi_{|R|}\}$.
- In some contexts we write V to denote the set $V_T \cup V_N$ (sometimes called the vocabulary).

Letter usage conventions Through-out the text specific letters and symbols will conventionally represent members of certain sets from Definition 2.1.

1. Lower-case letters from the beginning of the alphabet **a**, **b**, **c**, ... digits **0**, **1**, **2**, ... operator symbols $+$, $-$, $*$, ... and other similar characters are terminal symbols – i.e. members of V_T.
2. Upper-case letters from the beginning of the alphabet A, B, C, ... are non-terminal symbols – i.e. members of V_N.
3. Lower-case letters from the end of the alphabet x, y, z, w, ... are strings of terminal symbols – i.e. members of V_T^*.
4. Lower-case Greek letters like α, β, γ, ... are strings of symbols (both terminal or non-terminal) – i.e. members of $V^* = (V_T \cup V_N)^*$
5. Upper-case letters from the end of the alphabet X, Y , ... are members of V

Definition 2.2. Given a context-free grammar $G = (V_N, V_T, S, R)$ and two strings $\alpha, \beta \in V^*$ we say that α *directly derives* to β (according to grammar G) and write $\alpha \Rightarrow \beta$ iff there exist some $\gamma_0, \gamma_1 \in V^*$ and a production rule $(A \to \delta) \in R$ such that $\alpha = \gamma_0 A \gamma_1$ and $\beta = \gamma_0 \delta \gamma_1$.

Note. Another interpretation of Definition 2.2 is that every context-free grammar G is associated with a direct derive relation between strings of symbols

$$\Rightarrow \subseteq V^* \times V^*$$

Definition 2.3. A *derivation* from α to β (according to grammar G) is a sequence of strings $\gamma_0, \ldots, \gamma_n \in V^*$, $n \geq 0$ such that $\gamma_0 = \alpha$, $\gamma_n = \beta$ and $\gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n$ (for each $i = 1, \ldots, n$ string γ_{i-1} directly derives to string γ_{i+1} by some production rule $A_i \to \alpha_i$). A special zero-length derivation occurs when $\alpha = \beta = \gamma_0$.

The following additional notation is related to the definition of derivation:

1. We write $\alpha \overset{*}{\Rightarrow} \beta$ iff there exists a derivation from α to β .
2. We write $\alpha \overset{k}{\Rightarrow} \beta$ for $k \geq 0$ iff there exists a derivation from α to β of length k.
3. We use symbol Δ to denote a concrete derivation, i.e., $\Delta = (\gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n)$.
4. If Δ is a derivation from α to β then $\beta$is called a *yield* of the derivation and it is denoted as $y(\Delta)$. We say that derivation Δ yields string β.

Note. We may also define relation $\overset{*}{\Rightarrow} \subseteq V^* \times V^*$ as a transitive and reflexive closure of the direct derive relation.

Derivations from the start symbol produce strings that have special role. These strings are called "sentential forms". The non-rewritable sentential forms consisting only of terminal symbols are called "sentences" and they form the language generated by the grammar.

Definition 2.4. If $G = (V_N, V_T, S, R)$ is a context-free grammar then every string $\alpha \in V^*$ for which $S \overset{*}{\Rightarrow} \alpha$ is a *sentential form.*

Definition 2.5. Language L_G generated by a context-free grammar $G = (V_N, V_T, S, R)$ is defined as

$$L_G = \left\{ x \mid S \overset{*}{\Rightarrow} x \wedge x \in V_T^* \right\}$$

.

Definition 2.6. Let $G = (V_N, V_T, S, R)$ be a context-free grammar.

1. We say that symbol $X \in V$ is *generating* if $X \overset{*}{\Rightarrow} w$ for some $w \in V_T^*$.
2. We say that symbol $X \in V$ is *reachable* if $S \overset{*}{\Rightarrow} \alpha X \beta$ for some $\alpha, \beta \in V^*$.

Definition 2.7. Context-free grammar G is in a *normalized form* if it does not contain any unreachable and non-generating symbols.

Theorem 2.1. *Let $G = (V_N, V_T, S, R)$ be a context-free grammar such that $L(G) \neq \emptyset$. Let $G' = \left(V_N', V_T', S, R'\right)$ be the grammar obtained from G by first removing all non-generating symbols and production rules involving these symbols and second removing all unreachable symbols. Then G' is in a normalized form and $L(G') = L(G)$.*

Theorem 2.1 shows that every context-free grammar can be easily transformed into normalized form. Therefore we can assume without loss of generality that all grammars are in the normalized form unless explicitly stated otherwise. This will simplify our discussion of sentence generation algorithms. For the proof of Theorem 2.1 and a more detailed discussion of grammar transformations see [19].

2.2 Programming Languages

Programming languages are artificial languages used as systems of notation for describing computation [39]. Unlike natural languages, programming languages are formal enough to allow the described computation to be performed by a machine. Formal description of a computation expressed in a programming language is called a *program*. Programs usually have a textual representation.

One of the primary purposes of a programming language is to provide abstraction over the underlying computational model, i.e. to abstract over the construction details of a concrete machine that executes programs. Thus the same program can be executed on different types of hardware. Because of this abstraction programs are usually not executable directly. In order to be executed, a program has to be transformed from its textual representation (also called the *source code*) into a sequence of machine-dependent instructions also known as the "machine code". These transformations are performed by a class of software tools called *compilers*.

To allow human users (programmers) to write programs and also to allow implementation of any language processing tools (particularly compilers) a precise description of the programming language is needed. The normative description of a programming language is called *specification*. For most programming languages the specification is given in a semi-formal way in form of language reference manual (LRM).

The specification of a programming language traditionally differentiates between *syntax* - the form of programs and *semantics* - the meaning of programs. Syntax defines which strings of symbols are valid programs. Semantics assigns meaning to the valid programs. This distinction is sometimes blurred in the field of compiler construction when syntax and semantics interoperate in several phases of the compiler (see Section **??**).

Programs, seen as sequences of symbols, have compositional structure. This mean that program as a whole can be explained in terms of its parts (constructs), which in turn can be explained in terms of their subparts (subconstructs). The structure and form of programs are defined by the syntax

Class	*Language / Grammar*	*Automaton*
$\mathcal{L}_0$	Recursively enumerable	Turing machine
$\mathcal{L}_1$	Context-sensitive	Linear-bounded Turing machine
$\mathcal{L}_2$	Context-free	Pushdown automaton
$\mathcal{L}_3$	Regular	Finite state automaton

Table 2.1: Chomsky hierarchy of formal languages

of a programming language, i.e. syntax determines which strings of symbols are valid programs.

2.2.1 Syntax

Syntax of a programming language describes valid programs. Formally, programs can be seen as strings of symbols from a given alphabet Σ. In this view syntax defines programming language as formal language

$$L \subseteq \Sigma^*$$

Formal languages can be classified according to their expressive power by the Chomsky hierarchy as shown in Table 2.1. Each level of the hierarchy describes a class of formal languages going from $\mathcal{L}_3$ to $\mathcal{L}_0$ each class being a proper subset of the previous one. Chomsky showed that each of the classes is characterized by a specific type of formal grammars.

For theoretical reasoning programming languages are also considered as formal languages. As such they usually stand very close to class $\mathcal{L}_2$. However, most of them still contain features that can not be described by context-free grammars. This poses a question how to formally characterize a programming language.

It is not feasible to work with grammars more expressive than $\mathcal{L}_2$. When we start exploring the higher levels of Chomsky hierarchy, we encounter a steep increase in complexity of even basic grammar operations. For example, the membership problem is undecidable for context-sensitive languages in general . There are even some context-sensitive languages where the membership problem is undecidable for a fixed language.

For these reasons programming languages are usually defined in two steps:

1. First a suitable context-free approximation of a given programming language is chosen[2]. Because of the traditional compositional struc-

[2]A context-free approximation is a language that belongs to class is $\mathcal{L}_2$ and is a superset of the given programming language (i.e. it also contains strings that are not valid programs).

ture, most programming languages are essentially context-free in most aspects. Therefore it is usually possible to find a good approximation[3] of the language by a context-free grammar. This grammar is called the *context-free grammar of the programming language*. Some key non-terminal symbols of this grammar tend to coincide with important concepts of the programming language like declarations, statements, expressions, types, etc. They are sometimes called *syntactic categories*.

2. In the second step, the context-free grammar of the programming language is augmented with a set of constraints which rule out the remaining invalid programs. We will call these constraints *context-sensitive rules*. Context-sensitive rules are usually expressed in terms of the sentence parse structure according to the underlying context-free grammar (i.e. they are not independent from the syntactic categories defined by context-free grammar of the programming language).

Most important types of context-sensitive rules shared by many programming language are discussed in the next section.

Backus-Naur Form The most commonly used notation for expressing the context-free grammar of a programming language is Backus-Naur From (BNF). This notation is used almost ubiquitously in programming language specifications and language reference manuals. [4]. Distinctive features of the BNF notation as summarized by Knuth in [22] are the following:

1. Non-terminal symbols are distinguished from terminal symbols by enclosing them in special brackets.

2. All production rewriting a specific non-terminal are grouped together (i.e. a system of production rules $\{A \to BC, A \to \mathbf{d}, A \to C\}$ would be written as "$\langle A \rangle ::= \langle B \rangle \, \langle C \rangle \mid \mathbf{d} \mid \langle C \rangle$".

3. The symbol "::=" is used to separate the non-terminal on the left from the set of alternatives on the right.

4. The symbol "|" is used to separate alternatives.

5. Full names indicating the meaning of the strings being defined are used for non-terminal symbols.

[3] A superset language is a good approximation if it does not contain "too much" additional strings.

[4] In addition to the original variant proposed by J. Backus and P. Naur there are several extended variants of BNF usually abbreviated as EBNF.

$$
\begin{aligned}
\langle expression \rangle &::= \langle expression \rangle + \langle expression \rangle \\
&\mid \langle expression \rangle - \langle expression \rangle \\
&\mid \langle number \rangle \\
\\
\langle number \rangle &::= \langle digit \rangle \mid \langle digit \rangle \langle number \rangle \\
\\
\langle digit \rangle &::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}
\end{aligned}
$$

Figure 2.1: Simple grammar for numeric expressions expressed in BNF

In our view the point 5. seems to be the most important. It leads to usage of proper names for syntactic categories in the grammar and thus greatly simplifies the formulation of context-sensitive rules (which frequently refer to the grammatical structure of the programs). An example of a context-free grammar written in BNF is shown on Figure 2.1. For a more detailed discussion of BNF and grammar notations please refer to [22].

2.2.2 Context-sensitive Syntax

We use the term *context-sensitive syntax* in reference to the set of context-sensitive rules as they were described in the previous section[5].

The classification in the Chomsky hierarchy is not the only distinctive feature of the context-sensitive rules. Even though these rules define only the form of the programs (i.e. to rule out invalid programs), they often refer to the syntactic elements of programs according to their meaning. Hence the context-sensitive syntax is sometimes called *static semantics* in the literature . The following quotation from the `comp.compilers` newsgroup concisely summarizes different meaning of syntax.

> "Syntax is to do with structure and can be checked by static analysis of the program. We can then distinguish context free syntax which is to do with raw structure from context sensitive syntax which is to do with consistency within raw structure. We can also distinguish concrete syntax which is to do with representation from abstract syntax which is to do with meaningful structure.

[5]The term context-sensitive syntax was used by Tennent in [39] to distinguish the syntactic rules that ca not be expressed using a context-free grammar.

> Semantics is to do with meaning alone." – Greg Michaelson, 1987, *comp.compilers newsgroup*

The rules of context-sensitive syntax are usually specified informally (or semi-formally) by natural language descriptions. Consider an example from the Pascal programming language ISO/IEC 7185:1990 specification [20]:

> "The values denoted by the case-constants of the case-constant-lists of the case-list-elements of a case-statement shall be distinct (...)"

and the related grammar rules

$$\begin{aligned}
\langle \textit{case-statement} \rangle &::= \textbf{case}\ \langle \textit{case-index} \rangle\ \textbf{of}\ \langle \textit{case-list-element} \rangle \\
&\quad \{\ \textbf{;}\ \langle \textit{case-list-element} \rangle\ \}\ [\ \textbf{;}\ ]\ \textbf{end} \\
\langle \textit{case-list-element} \rangle &::= \langle \textit{case-constant-list} \rangle\ \textbf{:}\ \langle \textit{statement} \rangle \\
\langle \textit{case-constant-list} \rangle &::= \langle \textit{case-constant} \rangle\ \{\ \textbf{,}\ \langle \textit{case-constant} \rangle\ \} \\
\langle \textit{case-constant} \rangle &::= \langle \textit{constant} \rangle
\end{aligned}$$

The example also demonstrates how context-sensitive rules refer to the parse structure given by context-free grammar of the programming language.

The most important context-sensitive rules are related to the concepts of *declaration*, *scope* and *type system*. Therefore we will briefly introduce these concepts in the subsequent sections.

2.2.3 Type Systems

There is a large body of research on advanced type systems that usually appear in functional programming languages [35]. As this work focuses mainly on imperative programming languages[6] like Pascal and C, our discussion of type systems revolves around these languages.

In imperative programming languages, computation consists of a sequence of statements which access and modify state of the program. In a slightly simplified view, state of the program is represented by the contents of concrete memory regions. Variables provide an abstraction over these memory regions. They can hold different kinds of values ranging from simple logical values (true, false) to numeric values (integer or real number), character

[6] Also known as procedural programming languages as opposed to functional or logical programming languages.

strings and even memory addresses of other variables (pointers). Most operations like numeric addition or logical conjunction are meaningful only with certain kinds of values. Arithmetic operations require numeric values while logical operations require logical values. Typically it is an error to multiply a real number with a character string or to perform a logical negation of a memory address.

If the programming language does not keep track of the kinds of values that are stored in variables then they are treated uniformly as sequences of bits. However, one sequence of bits can be interpreted as different values depending on the context in which the variable is accessed (e.g. as a numeric value in an arithmetic operation or as a character string in some other operation). Such behavior is almost always unintentional and in practice leads to a large number of programming errors. Attempt to perform an operation with a misinterpreted value might cause the program to fail immediately. While this is certainly undesirable, in some cases a more serious failure comes into play: The operation proceeds and produces a meaningless or otherwise undefined value. Such erroneous value can potentially remain undetected for an arbitrarily long period of time and then suddenly cause a severe problem in a completely different part of the program. These kinds of problems are exceptionally hard to fix.

A common way to prevent typing errors is to equip the programming language with a *type system*. Type system defines what types of values exist in the programming language, what are their properties and what are the relationships between them. According to Pierce [35]

> "A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute."

A programming language with a type system is called a *typed programming language*. In such language every variable is associated with a *type*, which restricts the set of values that the variable can hold. Typing rules define the type of a result of an operation with respect to the types of the operands. Every programming language construct that returns a value is therefore associated with a type. The usual typing rules include rules like:

- The condition part of an `if-statement` must yield a value of a Boolean type.
- The operands of an arithmetic operator must have a numeric type (e.g. integer type or real type).

2.2.4 Declarations and Scopes

2.2.4.1 Declarations

In most programming languages, entities that appear in the program must be declared before they can be referred to. *Declarations* are programming language constructs that establish a binding between an identifier and an entity such as constant, variable, type or function. After such binding is established by a declaration, each subsequent occurrence of the given identifier in the program text constitutes a reference to the same entity. Rules that require agreement between declarations and entity references are the most common example of context-sensitive syntax.

Entity might not be fully specified in one declaration. A partial declaration of an entity might be completed by subsequent declarations. Some programming languages allow to write a function declaration that specifies only the function header (parameters and return type) and omits the function body. In C++ it possible to declare a class just by giving it a name (identifier). Declaration that fully specifies the entity is sometimes called a *definition*, e.g. function definition, constant definition, type definition. Entity can not have more than one definition.

2.2.4.2 Scopes

According to [15], the *scope* of a declaration is the region of the program within which the entity declared by the declaration can be referred to. A declaration is said to be in scope at a particular point in the program if and only if the declaration's scope includes that point. Scopes allow the same identifier to be bound to different entities at different points in the program.

In imperative programming languages a delimited sequence of statements (usually enclosed in parenthesis) is called a *block*. Blocks have to be properly nested, i.e. a nested block always ends before the parent block. Usually each block constitutes a separate scope. This technique is called lexical scoping.

Chapter 3

Problem Definition

This work attempts to automate the construction of test suites for compilers and other source code processing tools[1]. In our context, test suite is basically a collection of test inputs and outputs. Each input/output pair is called a test case. The testing consists of executing the software on every test input and comparing it's output with the expected output. When the software under test is a compiler, the test inputs have form of sample source programs. In this case it is very hard to determine if the output produced for the given source program is correct. Any target program semantically equivalent with the source program should be considered as correct output. In addition there are several ways how semantic equivalence could be defined. Although some authors have tried to solve these problems, we consider them to be out of scope of our work. When a compiler is executed on a test input we are interested only in the outcome of the compilation. If the compiler crashes or terminates with a compile error the test case is marked as failed. If the sample source program compiles successfully the test case is marked as passed. As a result, the test suites contain only test inputs and these are in form of sample source programs.

In this work we focus mainly on development of methods for automatic generation of sample source programs. At the abstract level all such methods are trying to solve the same problem, formally defined as sentence generation problem.

[1] In the rest of the text we talk almost exclusively about compilers since they present the most complicated problems. However, it is important to notice that the results of this work are applicable for all tools which process source code as their input, e.g. pretty-printers, refactoring engines, static analysis tools and others.

3.1 Sentence Generation Problem

Definition 3.1. Let L be a formal language over alphabet Σ. The problem of constructing a finite set $S \subset (\Sigma)^*$ of strings $w_1, w_2, \ldots, w_n$ such that

$$\forall i \in 1, \ldots, n : w_i \in L$$

is called the *sentence generation problem* for language L.

We attempt to solve the sentence generation problem for programming languages. For this purpose, programming languages are treated as formal languages – see Section 2.2.1. Programs are in turn considered as strings of symbols[2] that belong to the given formal language. The syntactic validity of programs is of primary importance and the meaning of programs is usually not considered at all. Semantic categories are involved to a limited extent, usually only to determine whether the advanced context-sensitive syntax rules are satisfied[3].

Since programming languages are infinite, we can not test compilers on every possible input. As Definition 3.1 suggests, we need to choose a finite subset of all inputs to be used for testing. There are several ways how to choose which sentences should be generated.

3.1.1 Randomized Sentence Generation

We have chosen to seek a solution to the sentence generation problem in a form of a randomized algorithm. In a randomized algorithm, some of the decisions are based on the values of random bits taken from an "external" source of randomness (in practice a pseudo-random number generator). A sentence generator based on this approach produces strings from a given language at random. In other words, the resulting sentence is determined by the values of random bits used by the generator. However, notice that the probability distribution of generated sentences over the whole language is heavily influenced by the design of the sentence generation algorithm.

3.1.2 Systematic Sentence Generation

An alternative to random sentence generation is systematic sentence generation. The idea is to generate strings from the given language systematically according to some deterministic procedure. The procedure can be presented

[2] We use terms *string of symbols*, *sequence of symbols* and *sentence* interchangeably.
[3] See Section 4.5 for more details.

in different forms. One example is the so-called *bounded-exhaustive* test data generation method [13, 38, 16]. The method is exhaustive in a sense that it systematically generates all strings from the given part of the language (or even from the whole language). To prevent extensively large or potentially infinite sets of test data it allows to define certain bounds on the generated strings. In other words, the method produces all strings within the given bounds.

A similar approach to systematic sentence generation called *controllable combinatorial coverage* was presented by Lämmel and Schulte in [25]. The authors limit the "combinatorial explosion" using explicit control mechanisms embedded in the context-free grammar that defines the target formal language. The mechanisms include

- Depth control – limit derivation depth; not just for the root nonterminal.
- Recursion control – limit nested applications of recursive production rules.
- Balance control – limit derivation depth variation between sibling nonterminals.
- Dependence control – limit combinatorial exhaustion of argument domains.

and also custom user-defined controls. Their implementation of controllable combinatorial coverage called *Geno* was used to generate test-data for problem-specific grammars of XPath, XML Schema, Web Services Policy Framework and others.

Another example can be found in the work of Daniel et al. [13] where a variant of bounded-exhaustive generation is used to generate test cases for refactoring engines.

3.2 Criteria for Generated Source Code

Compilers are usually tested using hand-written sets of programs that are created systematically to cover all constructs of the programming language one-by-one. However, such test suites are unable to cover the interaction of constructs in larger programs. Larger programs cause a combinatorial explosion of possibilities how individual language constructs can interact with one another.

The goal of automatic source code generation methods is to produce a test suite composed of sample programs which together exercise a significant portion of the given programming language. It is based on a hypothesis that testing compilers with such test suite will reveal a significant portion of software errors. We suggest that a large number of random programs can be expected to cover most of the interesting constructs of the given programming language and even their interactions. Therefore randomized sentence generation methods are well-suited for solving the sentence generation problem in our context. This idea is supported by other works on random testing such as [41, 2, 4, 17].

As mentioned before, the probability distribution of generated sentences is determined by the applied randomized sentence generation algorithm. To achieve the stated goals, our source code generation methods attempt to achieve probability distributions with certain properties. These properties are called *criteria for generated source code*. The fulfillment of all the criteria is not a strict requirement. The criteria are regarded more as a design principles and most of them are defined informally.

Correctness Criterion Generated sentences should be *valid*[4], self-contained programs in the given programming language. Here we refer to compiler-time correctness of programs. We do not require run-time correctness of programs, i.e. a generated program can (and frequently will) cause a run-time error such as division by zero or invalid memory access.

Coverage Criterion Every valid program should be generated with a non-zero probability. This criterion guarantees that testing with a generated set of programs is not limited to any arbitrary subset of the programming language. Alternatively it says that every defect that is discoverable through testing could be potentially discovered using the randomized sentence generation method. However, it is hard to prove that a source code generator fulfills this criterion in practice, i.e. a concrete implementation. Therefore it is applied more as a design principle.

Uniformity Criterion The probability distribution should attempt to achieve uniform distribution of programs with respect to a given metric. There are several meaningful metrics that can be used. For example: size of the program, structural complexity of the program, derivation length with respect to the underlying context-free grammar, derivation tree depth, etc. Generally a higher value of a metric means a more complicated program.

[4] Conforming to normative reference of the target language - e.g. ISO 7185 for Pascal

Because programming languages are infinite, we can not ask for uniform distribution over the whole language. As an illustration, it would mean that for any constant $n \in \mathbb{N}$ the aggregated probability of all programs of size smaller than n would be infinitely small. A probability distribution is uniform with respect to a given metric if all programs with the same value of the metric (e.g. same size) have the same probability. In a sensible distribution, the probability of a program would be inversely proportional to the value of its metric.

Again it is not easy to precisely control the probability distributions in a randomized sentence generation algorithm. Therefore the uniformity criterion is mostly used as guideline.

Note. Booth and Thompson showed in [5] that it is not possible to achieve arbitrary probability distribution over the strings of a context-free language when probabilities are assigned by a stochastic context-free grammar. Hence, depending of the grammar, it might be also impossible to achieve *uniform* probability distribution.

Regularity Criterion The regularity criterion asks for "regular" structure of programs. Program has a regular structure if its composed from elements with roughly equal size and complexity, i.e. the structural complexity should be evenly distributed among the program. Regular structure is an informal notion. A concrete requirement could be for example that the derivation tree of the program should be balanced (or as close to balanced as possible). Generally the criterion implies that degenerated programs should not be prevalent in the produced set of sample test programs. However, it is still useful to test with degenerated programs as they often present special cases that are quite likely to expose compiler defects.

Practicality Criterion Generated sentences should have reasonable size and complexity. They should generally resemble real-world programs. The set of test data for any kind of testing should be an approximation of possible real input data. In some cases[5], it makes sense to relax this criterion and generate huge programs with complexity orders of magnitude higher then typical program.

[5]For example stress testing or testing the boundary limits of the compiler.

3.3 Universal Source Code Generator

After we have discussed the desired properties of the outputs of source code generators, we turn our attention to the inputs.

Definition 3.2. *Universal source code generator* for a class C of programming languages[6] is a tool capable of generating sentences that belong to language L for any given $L \in C$.

As opposed to language-specific source code generators which are tailored for specific programming languages, universal generator takes the specification of a target programming language as input. The specification therefore has to be given in a well-defined representation that could be effectively processed by the generator. This requires a formalism that is expressive enough to define programming languages from the given class C and also could be encoded to a suitable representation on a computer. Finding such a formalism turns out to be a difficult problem.

Specifications of most real-world programming languages are written in a semi-formal way using combinations of BNF (see Section 2.2.1) and natural language[7]. In fact, there are only a few programming languages that have a formal specifications available. In addition, each of these few specifications uses a different incompatible formalism that is usually based on a special mathematical notation. Not only is it hard to represent such notation unambiguously on a computer, it is even harder to reason about it automatically with a computer program (such as the potential source code generator).

All these problems lead us to conclusion that the initial hope for a universal solution applicable to all programing languages is over-ambitious for the scope of this work. Therefore we will mainly concentrate on solutions targeting concrete programming languages. However, all sentence generation methods based on context-free grammars developed in Chapter 4 are universal if we consider the grammar as the specification of the target language. Unfortunately, as discussed in Section 2.2.1, context-free grammars are not expressive enough to serve as full specifications of real-world programming languages. These results can be used as a starting point for future work on source code generators.

[6] It would be ambiguous to talk about universal generator for *all* programming languages since there is no general definition of a programming language. Therefore we define universal generator for some class C of programming languages (e.g. languages expressible in a given formalism).

[7] Some quickly evolving languages like Ruby, Groovy, PHP, etc. do not have a definitive specification at all.

Target Programming Languages In this work we focus mainly on *imperative* procedural languages. Canonical examples are provided by languages Pascal and C. We have also performed some research to apply the current results to more advanced classes of programming languages, namely *functional* and *object-oriented* programming languages. We have used Haskell and SML to represent functional languages and Java, C++ to represent object-oriented languages. The advanced functional and object-oriented languages share the basic concepts (e.g. types, declarations and expressions) with the simpler procedural languages. As far as these similarities go, we can directly apply the source code generation methods described in this work. Specific methods that would deal with the additional features of functional and object-oriented languages are out of the scope of this work.

Chapter 4

Solution

Formal grammars have many applications in different fields of computer science. They are a very useful tool for describing formal languages – programming languages in particular[1]. In this work, grammars provide a conceptual underpinning of our approach to source code generation.

4.1 Abstract Sentence Generation Algorithm

This chapter describes a solution to the sentence generation problem based on context-free grammars. An abstract sentence generation framework is developed by transforming the definition of context-free derivation (see Definition 2.3) into a constructive algorithmic process.

If $G = (V_N, V_T, S, R)$ is a context-free grammar in a normalized form according to Definition 2.7, then every valid string from the language $L(G)$ can be generated by the *abstract sentence generation algorithm* (Algorithm 4.1). When the algorithm terminates and returns a string w as a result (i.e. $w = \gamma_n$) then w is such that

$$S \overset{*}{\Rightarrow} w \ \wedge \ w \in V_T^*$$

Hence by Definition 2.5 $w \in L_G$. Every string produced by the algorithm therefore belongs to the language L_G. Conversely every derivation $S \overset{*}{\Rightarrow} x$ of a terminal string $x \in V_T^*$ according to the grammar G corresponds to a specific computation of the algorithm. For every string $x \in L_G$ there is a derivation $S \overset{*}{\Rightarrow} x$ and therefore every string $x \in L_G$ can be returned (computed) by the algorithm.

[1] Programming language parsers are usually constructed from grammar-based descriptions. Different types of grammars can be used as models of computation (see Section 2.2.1). Grammars are also important in the field of computational linguistics.

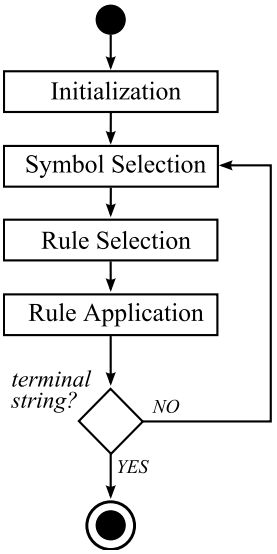


Figure 4.1: Structure of the sentence generation algorithms

Algorithm 4.1 Abstract sentence generation algorithm

1. Assign $\gamma_0 = S$ and $i = 0$ (γ_0 is a string consisting of the start symbol of grammar G).

2. For the given string γ_i (starting from $i = 0$) choose a non-terminal $A \in \gamma_i$ (i.e. $\gamma_i = \delta_0 A \delta_1$ for some strings δ_0, δ_1).

3. For the non-terminal A choose a production rule $(A \rightarrow \alpha) \in R$.

4. Compute new string $\gamma_{i+1} = \delta_0 \alpha \delta_1$ by applying the rule $A \rightarrow \alpha$ to γ_i.

5. If γ_{i+1} does not contain any more non-terminal symbols return γ_{i+1} as a result otherwise continue with Step 2 substituting $i \mapsto i + 1$.

Note. There is always at least one non-terminal to choose from in Step 2 since the algorithm starts with $\gamma_0 = S$ and terminates in Step 5 if there are no more non-terminals symbols.

Note. There is always at least one rule to choose from in Step 3 because we assume that grammar G is in the normalized form (i.e. it does not contain non-terminals that can not be further rewritten).

Algorithm 4.1 is called "abstract" since it leaves two important questions open:

- Which non-terminal to choose in Step 2?
- Which production rule to choose in Step 3?

There are several ways to answer both of these questions. Depending on how these questions are answered, different sentence generation algorithms are obtained. In this sense the abstract sentence generation algorithm provides a framework for applying concrete symbol selection and rules selection methods. These methods are discussed in the subsequent sections.

Termination Problem When arbitrary choices of symbols and production rules are allowed, the abstract sentence generation algorithm does not guarantee finite termination. For example, consider a grammar $G_{simple} = (V_N, V_T, R, S)$ where $V_N = \{S\}$, $V_T = \{\mathbf{a}\}$ and R contains production rules

$$\begin{array}{lcl} S & \rightarrow & SS \\ S & \rightarrow & \mathbf{a} \end{array}$$

If the algorithm always chooses to apply the first rule $S \rightarrow SS$ in Step 3, it produces an infinite sequence of strings $(\gamma_i)_{i=0}^{\infty}$ that looks like

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow \cdots$$

In such case the algorithm never terminates. The possibility of infinite generation naturally poses a practical problem. We will refer to it as *termination problem*. For each of the concrete methods developed in subsequent sections we analyze its impact on the termination problem.

4.2 Symbol Selection Methods

4.2.1 Depth-first Symbol Selection

In terms of the abstract sentence generation algorithm, the point of depth-first symbol selection (DFSS) method is to always choose the left-most non-terminal in the symbol selection step (see Algorithm 4.2).

The unique features of this method are better understood when we picture the sentence generation process as a process of derivation tree construction (as opposed to string rewriting).

Algorithm 4.2 Sentence generation algorithm with DFSS method

1. Put $\gamma_0 = S$ and $i = 0$.
2. Find the non-terminal $A \in \gamma_i$ such that $\gamma_i = xA\delta$ for some $x \in V_T^*$, $\delta \in (V_T \cup V_N)^*$ (i.e. the left-most non-terminal in γ_i).
3. Choose a production rule $(A \to \alpha) \in R$.
4. Put $\gamma_{i+1} = x\alpha\delta$ (application of the rule $A \to \alpha$ to γ_i).
5. If γ_{i+1} does not contain any more non-terminal symbols return γ_{i+1} as the result otherwise put $i \mapsto i + 1$ and continue with Step 2 .

$$
\begin{aligned}
E &\to E + E \\
E &\to E - E \\
E &\to E * E \\
E &\to E / E \\
E &\to (\, E \,) \\
E &\to \mathbf{id}
\end{aligned}
$$

Figure 4.2: Sample grammar for arithmetic expressions

Definition 4.1. Let $G = (V_N, V_T, S, R)$ be a context-free grammar. A *derivation tree* according to G is a tree such that

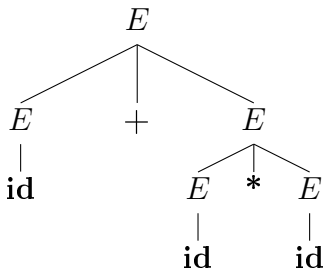
1. Each node of the tree is labeled with some symbol from $V_N \cup V_T$ or with a special symbol ϵ. Inner nodes are labeled with symbols from V_N. Leaf nodes are labeled with terminal symbols or with ϵ.
2. Root of the tree is labeled with symbol S.
3. Inner node labeled with $A \in V_N$ has child nodes labeled in order from left to right with symbols $X_1, \ldots, X_k$ only if there is a production rule $A \to X_1 X_2 \cdots X_k$ in R.
4. Leaf node can be labeled with ϵ only if it's the sole child of a parent node labeled with $A \in V_N$ and there is a production rule $A \to \epsilon$ in R.

Every execution of the abstract sentence generation algorithm corresponds to a concrete derivation according to the given grammar. In turn,

every derivation corresponds to some derivation tree. A derivation tree corresponding to a derivation $S \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n$ can be constructed one step at a time starting with the root node labeled with S. Each step of the derivation (i.e. each application of a production rule) is expressed by adding child nodes to a tree node in the derivation tree. The child nodes are labeled according to the body of the applied production rule and the parent node corresponds to the expanded non-terminal. Instead of replacing a symbol with a substring we are simply replacing a node with a subtree. If we take the grammar shown in Figure 4.2 as an example, the derivation

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow \\ &\Rightarrow \mathbf{id} + E \Rightarrow \\ &\Rightarrow \mathbf{id} + E * E \Rightarrow \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \Rightarrow \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

would corresponds to the derivation tree



Depth-first symbol selection method can be seen as analogous to depth-first search on the derivation tree. The nodes of the derivation tree are created in the same order as they would have been traversed by the standard depth-first search algorithm[12]. This behavior has several implications. In particular when the algorithm enters a subtree it will not leave until all nodes of the subtree are fully expanded, i.e. until all leaf nodes in the subtree are labeled with terminal symbols. In another words the algorithm always fully expands the given non-terminal symbol until moving on to the next symbol – corresponding to next sibling or parent. See Figure 4.3 for illustration of this property.

4.2.2 Breadth-first Symbol Selection

Breadth-first symbol selection (BFSS) is a non-terminal selection method that always selects the non-terminal with lowest depth in the derivation tree.

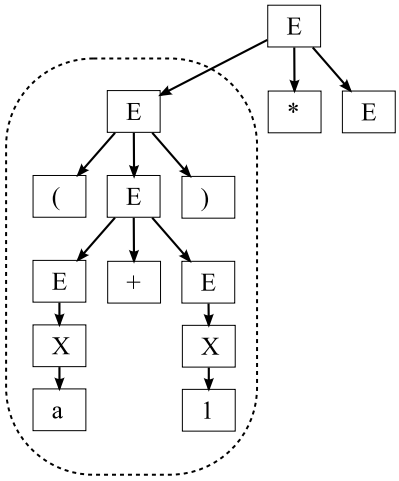


Figure 4.3: Derivation tree with a fully expanded subtree

This guarantees that all non-terminals on the given level of the derivation tree are expanded before progressing to the next level. Alternatively the method can be formulated as rewriting all non-terminals in parallel in each step (i.e. expanding whole level of the derivation tree in one step). An adaptation of the abstract sentence generation algorithm using this approach is presented in Algorithm 4.3.

When using the BFSS method, symbol nodes are created in the same order as they would have been discovered by the breadth-first traversal of the derivation tree.

The BFSS method has an important advantage if we want to place an upper bound on the size of the generated derivation tree, i.e., if we want to stop the generation process after a certain number of symbol nodes is produced. In comparison to DFSS it produces much more balanced derivation trees. DFSS fully expands given non-terminal before progressing to the next non-terminal on the same level of the derivation tree. When DFSS generation process is stopped at arbitrary point, some branches of the derivation tree are likely to be degenerated, i.e. completely unexpanded.

On the other hand, BFSS leads to complications when dealing with context-sensitive syntax rules. Related issues are discussed in Subsection 4.5.2.1 and Subsection 4.5.2.2.

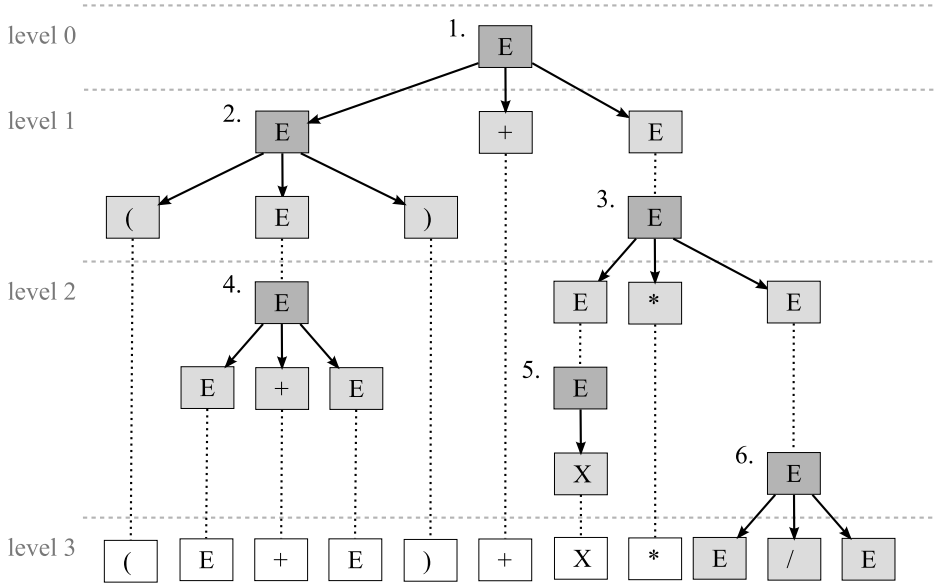


Figure 4.4: Example of step-by-step breadth-first derivation

Algorithm 4.3 Breadth-first generation algorithm

1. Assign $\gamma_0 = S$ and $i = 0$.
2. Let k be the number of non-terminals in γ_i. Find non-terminals $A_1, \ldots, A_k$ such that $\gamma_i = x_0 A_1 x_1 \cdots x_{k-1} A_k x_k$ for some $x_0, x_1, \ldots, x_k \in V_T^*$ (i.e. all non-terminals on the i-th level of the derivation tree).
3. For all $j = 1, \ldots, k$ choose a production rule $(A_j \to \alpha_j) \in R$.
4. Put $\gamma_{i+1} = x_0 \alpha_1 x_1 \cdots x_{k-1} \alpha_k x_k$ (parallel application of rules $A_j \to \alpha_j$ for $j = 1, \ldots, k$).
5. If γ_{i+1} does not contain any more non-terminal symbols return γ_{i+1} as the result otherwise put $i \mapsto i + 1$ and continue with Step 2 .

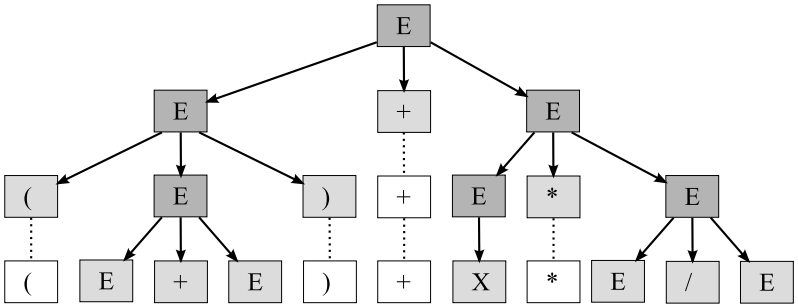


Figure 4.5: Example of breadth-first derivation – whole level expanded in each step.

4.3 Rule Selection Methods

4.3.1 Random Rule Selection

Random rule selection (RRS) is a straightforward rule selection method where production rules are chosen randomly. More precisely each rule is picked uniformly at random from a set of production rules associated with the non-terminal symbol chosen in the preceding symbol selection step. This is an obvious constraint implied by the definition of abstract sentence generation algorithm (see Section 4.1).

The major disadvantage of the RRS method is that it does not provide any means for dealing with the termination problem. As an illustration we show that the probability of termination when generating with RRS method according to the simple grammar for arithmetic expressions defined in Figure 4.2 is only 25%. The termination probability is computed as a result of the equation 4.1 that is derived from the structure of the context-free grammar.

$$\begin{aligned} p_t(E) &= \tfrac{1}{6}.(p_t(E))^2 \\ &+ \tfrac{1}{6}.(p_t(E))^2 \\ &+ \tfrac{1}{6}.(p_t(E))^2 \\ &+ \tfrac{1}{6}.(p_t(E))^2 \\ &+ \tfrac{1}{6}.p_t(E) \\ &+ \tfrac{1}{6}.p_t(\mathbf{id}) \end{aligned} \tag{4.1}$$

$$p_t(E) = \tfrac{4}{6}.(p_t(E))^2 + \tfrac{1}{6}.p_t(E) + \tfrac{1}{6}$$

$$p_t(E) = \tfrac{1}{4} = 25\%$$

In the equation, $p_t(A)$ denotes the probability that non-terminal A will yield a sequence of terminal symbols in finite number of steps. Since **id** is a terminal symbol $p_t(\mathbf{id}) = 1$. There are 6 production rules associated with the non-terminal E in the given grammar and hence the probability of each rule is $\frac{1}{6}$.

Our example with grammar for arithmetic expression represents a common phenomenon in programming languages. The high probability of non-terminating computation such as 75% is prohibitive for practical applicability of this method. This gives a strong motivation for developing other rule selection methods.

4.3.2 Probabilistic Rule Selection

Probabilistic rule selection (PRS) method is an attempt to enhance the RRS method to prevent or to reduce the probability of non-terminating computation. Production rules are still selected randomly, but with probability distributions that are not necessarily uniform. The possibility to change these probability distributions together with the input grammar gives us additional degree of control over the generation process.

The idea of specifying discrete probability distributions on selected sets of production rules is formalized in concept of stochastic context-free grammars[2].

4.3.2.1 Stochastic Context-free Grammars

Stochastic context-free grammars (SCFG) are well-known in the literature [3, 5, 7, 8, 9, 11, 10, 26, 34, 37, 40]. We define them as a special subset of more general weighted context-free grammars.

Weighted context-free grammars (WCFG) extend standard context-free grammars with non-negative numeric weights for all production rules.

Definition 4.2. *Weighted context-free grammar* G is a 5-tuple (V_N, V_T, S, R, ω) where

- $G_0 = (V_N, V_T, S, R)$ is a context-free grammar according to Definition 2.1,
- $\omega : R \to \mathbb{R}_0^+$ is a weight function assigning non-negative weight to each production rule from grammar G_0

[2] In other texts, stochastic context-free grammars are also referred to as *probabilistic* context-free grammars.

We write $\omega(A \to \alpha)$ to denote the weight of production rule $A \to \alpha$.

Stochastic context-free grammars can be considered as a special case of weighted context-free grammars where production rule weights satisfy additional constraints. These constraints allow the weights to be interpreted as discrete probabilities of individual production rules with respect to other production rules associated with the same non-terminal symbol.

Definition 4.3. *Stochastic context-free grammar* $G = (V_N, V_T, S, R, \omega)$ is a weighted context-free grammar with a weight function ω that has the following properties:

- $range(\omega) \subseteq \langle 0, 1 \rangle$ or equivalently $\omega : R \to \langle 0, 1 \rangle$
- probabilities of production rules for every non-terminal $A \in V_N$ sum up to 1, i.e.,
$$\forall A \in V_N : \sum_{(A \to u) \in R} \omega(A \to u) = 1$$

Our definitions of weighted and stochastic context-free grammars are equivalent to those found in the standard literature.

Every weighted context-free grammar G can be transformed into corresponding stochastic context-free grammar G'. The production rule probability distributions in the corresponding SCFG G' are proportional to weight distributions in G. This transformation is called *normalization* of WCFG.

Lemma 4.1. *There exists a mapping norm that maps every weighted context-free grammar* $G = (V_N, V_T, S, R, \omega)$ *to a stochastic context-free grammar* $G' = (V_N, V_T, S, R, \omega')$ *such that*

$$\forall (A \to u), (A \to v) \in R : \frac{\omega(A \to u)}{\omega(A \to v)} = \frac{\omega'(A \to u)}{\omega'(A \to v)} \tag{4.2}$$

Proof. Given a WCFG $G = (V_N, V_T, S, R, \omega)$, we show how to construct ω' satisfying property (4.2) such that $G' = (V_N, V_T, S, R, \omega')$ will be a SCFG according to Definition 4.3. Let ω' be a weight function defined for all $(A \to u) \in R$ as

$$\omega'(A \to u) = \frac{1}{\sigma_A} \cdot \omega(A \to u)$$

where σ_A is the sum of weights of all productions associated with non-terminal $A \in V_N$ defined as

$$\sigma_A = \sum_{(A \to u_o) \in R} \omega(A \to u_0)$$

Now for all $(A \to u), (A \to v) \in R$

$$\frac{\omega'(A \to u)}{\omega'(A \to v)} = \frac{\frac{1}{\sigma_A} \cdot \omega(A \to u)}{\frac{1}{\sigma_A} \cdot \omega(A \to v)} = \frac{\omega(A \to u)}{\omega(A \to v)}$$

hence ω' satisfies property (4.2). For all $(A \to u) \in R$

$$0 \leq \omega(A \to u) \leq \sigma_A$$

$$0 \leq \frac{1}{\sigma_A} \cdot \omega(A \to u) \leq 1$$

$$\omega'(A \to u) \in \langle 0, 1 \rangle \tag{4.3}$$

and also for all $A \in V_N$

$$\sum_{(A \to u) \in R} \omega'(A \to u) = \frac{1}{\sigma_A} \cdot \sum_{(A \to u) \in R} \omega(A \to u) = \frac{1}{\sigma_A} \cdot \sigma_A = 1 \tag{4.4}$$

Equations (4.3) and (4.4) show that ω' satisfies both properties from Definition 4.3 and therefore $G' = (V_N, V_T, S, R, \omega')$ is a stochastic context-free grammar. □

Note. Stochastic context-free grammars are extensively studied in the field of formal linguistics and natural language processing. See Section 6.2 for more details.

4.3.2.2 Probabilistic Sentence Generation Algorithm

The *probabilistic sentence generation algorithm* (Algorithm 4.4) is an instantiation of the abstract sentence generation algorithm using the PRS method. For illustration, a detailed description of the algorithm in given on Algorithm 4.4. Sentence generation is driven by a stochastic-context free grammar $G = (V_N, V_T, S, R, \omega)$.

The algorithm makes a pseudo-random decisions when choosing production rules in Step 3. Therefore it is a *randomized algorithm*. Computation of a randomized algorithm is not fully determined by it's input. It also depends on the values produced by a pseudo-random number generator [12]. In our case the input data have a form of grammar specification. Hence for a fixed grammar the computation of probabilistic sentence generation algorithm is fully determined by the seed number of the pseudo-random generator.

$$
\begin{array}{lll}
E & \to [1/8] & E + E \\
E & \to [1/8] & E - E \\
E & \to [1/8] & E * E \\
E & \to [1/8] & E/E \\
E & \to [1/6] & (\,E\,) \\
E & \to [1/3] & \mathbf{id}
\end{array}
$$

(a) first

$$
\begin{array}{lll}
E & \to [1/16] & E + E \\
E & \to [1/16] & E - E \\
E & \to [1/16] & E * E \\
E & \to [1/16] & E/E \\
E & \to [1/4] & (\,E\,) \\
E & \to [1/2] & \mathbf{id}
\end{array}
$$

(b) second

When the PRS method is applied the probability $p_t(E)$ of terminating computation is 66% for grammar a) and full 100% for grammar b). Compare this with 25% achieved by the RRS method on the same underlying context-free grammar as shown in the previous section.

Figure 4.6: Stochastic context-free grammars for generating expressions

Algorithm 4.4 Probabilistic sentence generation algorithm

1. Let $\gamma_0 = S$ and $i = 0$. [Initialization]
2. For the given string γ_i choose a non-terminal A such that $\gamma_i = \delta_0 A \delta_1$ for some strings δ_0, δ_1. [Symbol Selection]
3. Among rules rewriting non-terminal A choose one production rule $A \to \alpha$ *at random* with respect to the probability distribution given by $\omega\colon R \to [0, 1]$. [Rule Selection]
4. Compute new string $\gamma_{i+1} = \delta_0 \alpha \delta_1$ by rewriting non-terminal A in γ_i with the body of the rule $A \to \alpha$.
5. If γ_{i+1} does not contain any more non-terminal symbols return γ_{i+1} as a result otherwise continue with Step 2 substituting $i \mapsto i + 1$.

Running-time Analysis The analysis of the running time is performed differently for the randomized algorithms than for deterministic algorithms. In case of randomized algorithms the running time becomes a random variable and the analysis involves understanding the distribution of this random variable [32]. We are usually concerned with the *expected running time* rather that with the worst-case scenario. The worst-case running time typically occurs with a very small probability.

We apply this kind of analysis to the probabilistic sentence generation algorithm. When the algorithm generates strings according to $G = (V_N, V_T, S, R, \omega)$, the number of iterations is proportional to the length of the derivation it follows. Hence the expected running time of the algorithm is given by the probability distribution of the derivation lengths according to G. Usually there is no upper bound on the length of a derivation since all non-trivial grammars that we are interested in generate infinite languages. Intuitively, to achieve finite expected running time, the probability of a derivation has to be inversely proportional to it's length. The assignment of probabilities in stochastic context-free grammars is discussed in more detail in Subsection 4.4.1.

4.3.3 Finalization Rules

All rule selection methods presented so far face the termination problem to some extent. The length of the derivation (and hence also the length of derived string) can become too large and potentially even infinite. One way to deal with this problem is to set an explicit upper bound on the length of the generated sentence or on the number of derivation steps. The question is how to enforce such upper bound in the sentence generation process.

A straightforward idea is to stop the generation algorithm when the length of the intermediate sentential form reaches the given bound. However, the sentential form obtained in this way can still potentially contain unexpanded non-terminal symbols. The finalization rule selection (FRS) method tries to solve this problem.

The main point of the FRS method to choose production rules that lead to termination in the least possible amount of derivation steps. There rules are called *finalization rules*. To introduce finalization rules formally, we need a few intermediate definitions. Let $G = (V_N, V_T, S, R)$ be a fixed context-free grammar. Assume that a reference to G is implicit in subsequent definitions.

Definition 4.4. For every $k \geq 0$ and every non-terminal $A \in V_N$ we define

1. $D(A)$, the set of terminal strings derivable from A, as

$$D(A) = \left\{ x \in V_T^* \mid A \overset{*}{\Rightarrow} x \right\}$$

2. $D(A, k)$, the set of terminal strings derivable from A in k steps, as

$$D(A, k) = \left\{ x \in V_T^* \mid A \overset{k}{\Rightarrow} x \right\}$$

Definition 4.5. For every non-terminal $A \in V_N$ we define the length of the shortest derivation from A that yields a terminal string as

$$\mu(A) = \min \{ k \in \mathbb{N}_0 \mid D(A, k) \neq \emptyset \}$$

Now we are ready to define finalization rules.

Definition 4.6. For every non-terminal symbol $A \in V_N$ the set of finalization rules $Fin\,(A)$ is defined as

$$Fin(A) = \left\{ A \to \alpha \mid \exists x \in D(A, \mu(A)) : \alpha \overset{\mu(A)-1}{\Longrightarrow} x \right\}$$

In the definition of finalization rules $D(A, \mu(A))$ stands for the set of potential "target strings". By Definition 4.4 every derivation from A that has the minimal length $\mu(A)$ and yields a terminal strings has to yield a string from $D(A, \mu(A))$. These derivations are called *finalizing derivations*. The Definition 4.6 basically says that $A \to \alpha$ is a finalization rule only if it appears as the first step of some finalizing derivation. This is expressed as a requirement that terminal string $x \in D(A, \mu(A))$ must be derivable from α in $\mu(A) - 1$ steps.

Relationship with symbol selection methods In principle, the finalization rule selection method can be used with both DFSS and BFSS methods. However, the BFSS method produces interim sentential forms with higher structural regularity during it's computation. As a consequence, combination of the BFSS method with the finalization rule selection method could be considered more practical.

4.3.3.1 Finalization Rule Discovery Algorithm

The FRS methods requires the sets of finalization rules for all non-terminal symbols of the given context-free grammar to be precomputed before the generation process is started. This is solved by the *finalization rule discovery*

algorithm. The algorithm computes values of $\mu(A)$ for all non-terminal symbols of the input grammar. With these values available it is easy to identify the finalization rules according to the definition.

The finalization rule discovery algorithm uses the technique of relaxation [12]. For each non-terminal symbol $A \in V_N$, it maintains an attribute $e[A]$, which is an upper bound on the length of shortest derivation from A that yields a terminal string. We call $e[A]$ the shortest-derivation estimate. At the beginning, the shortest-derivation estimates are initialized by the following procedure.

INITIALIZE-ESTIMATES(G, e)

```
1  for each A ∈ V_N
2        do e[A] ← ∞
3  for each (A → α) ∈ R
4        do if α ∈ V_T^*
5              then e[A] ← 1
```

Production rules $A \rightarrow \alpha$ with $\alpha \in V_T^*$ can rewrite their associated non-terminals to terminal strings in a trivial one step derivation. Therefore every non-terminal symbol that appears in the head of such rule is initialized with $e[A] = 1$. All other shortest-derivation estimates are initially set to infinity.

The process of relaxing a rule $A \rightarrow \alpha$ involves testing whether we can improve the shortest-derivation estimate for non-terminal A found so far by using the rule as the first step of the derivation. This is the case if the current shortest-derivation estimate for α is smaller than the current estimate for A by more than one step. The shortest-derivation estimate for α is computed as the sum of shortest-derivation estimates of non-terminals that appear in α. The following pseudo-code performs a relaxation step for rule $A \rightarrow \alpha$.

RELAX(A, α)

```
1  newMin ← 1
2  for i = 1 to length[α]
3        do if α[i] ∈ V_N
4              then newMin ← newMin +e[α[i]]
5  if newMin ≤ e[A]
6     then e[A] ← newMin
```

The finalization rule discovery algorithm maintains a set S of non-terminal symbols whose final shortest-derivation lengths have been already determined. The algorithm repeatedly selects the non-terminal $A_0 \in V_N - S$ with the minimum shortest-derivation estimate, adds A_0 to S, and relaxes all production rules which contain A_0 in the body.

FINALIZATION-RULE-DISCOVERY(G)

```
1  INITIALIZE-ESTIMATES(G, e)
2  S ← ∅
3  while S ≠ V_N
4        do find A_0 ∈ V_N − S such that e[A_0] = min {e[A] | A ∈ V_N − S}
5           S ← S ∪ A_0
6           for each (A → α) ∈ R such that α contains A_0
7                 do RELAX(A, α)
```

Running-time Analysis The search for non-terminal $A_0$with minimum $e[A_0]$ on the line 4 can be implemented using a priority queue. The queue would maintain the set of non-terminal $V_N - S$, of non-terminals, keyed by their current shortest-derivation estimates. If a binary heap is used as the priority queue implementation, the EXTRACT-MIN operation would take $O(\log |V_N|)$. However, after every RELAX(A, α) call on the line 7 we potentially have to perform the DECREASE-KEY operation. In the binary heap this operation also takes $O(\log |V_N|)$. This is not a big problem, since even if we precompute the set of non-terminals that appear in the body of each production rule, the RELAX procedure still runs in $O(|V_N|)$ time. Each production rule is relaxed at most $O(|V_N|)$ times – once for every non-terminal that appears in its body. Therefore the rule relaxation loop on lines 6–7 takes total time $O(|R| \,.\, |V_N|^2)$. The INITIALIZE-ESITMATES procedure can be modified to run in linear time, but that will not have affect on the total running time of the algorithm that is $O(|R| \,.\, |V_N|^2)$.

Analysis of correctness Because the finalization rule discovery algorithm always chooses the non-terminal with minimum shortest-derivation estimate in $V_N - S$, we can say that it uses a greedy strategy. As pointed out in [12] greedy strategies do not always produce optimal results in general. However, the following theorem shows that the algorithm indeed computes optimal shortest-derivation lengths.

Theorem 4.2 (Correctness of FRD algorithm)**.** *For every context-free grammar $G = (V_N, V_T, S, R)$, finalization rule discovery algorithm terminates with $e[A] = \mu(A)$ for all non-terminal symbols $A \in V_N$.*

The proof of this theorem closely resembles the proof of correctness of Dijkstra's algorithm. See [12] for more details.

4.4 Grammar Analysis

4.4.1 Grammar Consistency

Every finite computation of the abstract sentence generation algorithm[3] corresponds with a concrete grammar derivation. The number of iterations performed by the algorithm is proportional to the length of the corresponding derivation. If the sentence generation algorithm uses PRS method driven by a stochastic context-free grammar $G = (V_N, V_T, S, R, \omega)$, probability of each computation is given by the probability of corresponding derivation according to the grammar G.

Definition 4.7. Let $G = (V_N, V_T, S, R, \omega)$ be a stochastic context-free grammar. The probability of a derivation $\Delta = (\gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n)$ is defined as

$$p(\Delta) = \prod_{i=0}^{n-1} \omega(A_i \to \alpha_i)$$

where $A_i \to \alpha_i$ is the production rule used in the i-th step of the derivation (i.e. γ_i directly derives to γ_{i+1} by application of the rule $A_i \to \alpha_i$ for all $i = 0, \ldots, n-1$).

It follows from the discussion in Section 4.1 that the sentence generation algorithm can potentially enter a derivation sequence that has a finite probability of never terminating . We are interested in properties of SCFG that could guarantee that the probability of such derivation sequences is zero.

Definition 4.8. The probability $p(x)$ of string x is defined as aggregate probability of all derivations that yield x. That is

$$p(x) = \sum_{y(\Delta)=x} p(\Delta)$$

To guarantee termination of the sentence generation algorithm (i.e. prevent the termination problem), the aggregate probability of all strings that belong to the language $L(G)$ must be 1. It implies that no probability mass is lost in the derivation sequences of infinite length. This leads to a concept of grammar consistency introduced by Booth and Thompson in [5] and further studied by other authors in [40, 37, 34].

[3] Here we refer to all concrete instantiations of the abstract sentence generation algorithm.

Definition 4.9. A stochastic context-free grammar G is *consistent* if

$$\sum_{x \in L(G)} p(x) = 1$$

In our context consistency of the stochastic grammar guarantees that the sentence generation algorithm based on a PRS method always terminates. Therefore we focus on methods for determining consistency of the given SCFG.

Definition 4.10. Let $G = (V_N, V_T, S, R, \omega)$ be a stochastic context-free grammar where $V_N = \{A_1, A_2, \ldots, A_n\}$. The *stochastic expectation matrix* $\mathbf{E}$ of grammar G is defined as

$$\mathbf{E} = \begin{bmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,n} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{n,1} & e_{n,2} & \cdots & e_{n,n} \end{bmatrix}$$

where $e_{i,j}$ is the expected number of occurrences of non-terminal A_j in a sentential form that directly derives from non-terminal A_i (i.e. derives in one step using a production rule of a form $A_i \rightarrow \alpha$). Formally

$$e_{i,j} = \sum_{A_i \rightarrow \alpha} \omega(A_i \rightarrow \alpha).\, |\alpha \backslash A_j|$$

where $|\alpha \backslash A_j|$ is the number of occurrences of A_j in α.

Definition 4.11. *Spectral radius* ρ of a square matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ is defined as

$$\rho(\mathbf{A}) = \max_{k=1,\ldots,n} |\lambda_k|$$

where $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the (complex or real) eigenvalues of matrix $\mathbf{A}$.

Theorem 4.3 (Booth and Thompson [5])**.** *Stochastic context-free grammar G is consistent if and only if the spectral radius of the stochastic expectation matrix of grammar G is less then 1. That is*

$$\rho(\mathbf{E}) < 1$$

The theorem allows us to determine whether a given stochastic grammar is consistent. The spectral radius of $\mathbf{E}$ can be computed directly from the eigenvalue decomposition. Alternatively a faster algorithm described by Wetherell in [40] can be used. The algorithm does not compute the value of the spectral radius. It only determines if the spectral radius of a given matrix is smaller than 1.

4.4.2 Modular Consistency

We've shown in the previous section how to determine whether a stochastic context-free grammar is consistent and therefore appropriate for sentence generation. However, when the assignment of production rule probabilities results in an inconsistent grammar, it is useful to be able to determine which particular set of probabilities causes the inconsistency. This is important especially when the probabilities are assigned manually. In this section we develop a method that allows us to localize the cause of inconsistency to a set of non-terminal symbols.

Definition 4.12. *Derivability graph* of a context-free grammar $G = (V_N, V_T, S, R)$ is a directed graph whose vertices are non-terminal symbols from V_N and vertices such that

1. there is a vertex v_A for every non-terminal symbol $A \in V_N$
2. there is a directed edge from v_A to v_B iff there is a production rule $(A \to \alpha) \in R$ such that $B \in \alpha$ (i.e. if non-terminal B is directly derivable from non-terminal A).

The main idea is to compute strongly connected components of the derivability graph and consider consistency of each component separately. The whole grammar is consistent if and only if all the components are consistent.

Definition 4.13. Set of non-terminal symbols C is called a *derivability component* of a context-free grammar $G = (V_N, V_T, S, R)$ if it is a maximal subset of V_N such that for every two non-terminals $A, B \in C$ there is a directed path from v_A to v_B in the derivability graph of G.

Derivability components correspond to strongly connected components of a derivability graph of G. Hence the grammar can be decomposed into derivability components $C_1, C_2, \ldots, C_m$ such that

$$\bigcup_{i=0}^{m} C_i = V_N$$

We will denote the mapping of non-terminals to their respective derivability components as $comp : V_N \to \mathcal{P}(V_N)$.

Each derivability component C induces its own grammar as a restriction of the original grammar $G = (V_N, V_T, S, R)$. We get the induced grammar G_C when we:

$$
\begin{array}{rcl}
\langle prog \rangle & ::= & \langle stmt \rangle \mid \langle stmt \rangle \, ; \langle prog \rangle \\
\\
\langle stmt \rangle & ::= & \langle if\text{-}stmt \rangle \\
& \mid & \langle while\text{-}stmt \rangle \\
\\
\langle if\text{-}stmt \rangle & ::= & \textbf{if } \langle expr \rangle \textbf{ then } \langle stmt \rangle \\
\\
\langle while\text{-}stmt \rangle & ::= & \textbf{while } \langle expr \rangle \textbf{ then } \langle stmt \rangle \\
\\
\langle expr \rangle & ::= & \langle expr \rangle + \langle term \rangle \\
& \mid & \langle expr \rangle - \langle term \rangle \\
& \mid & \langle term \rangle \\
\\
\langle term \rangle & ::= & \langle term \rangle * \langle factor \rangle \\
& \mid & \langle term \rangle \, / \, \langle factor \rangle \\
& \mid & \langle factor \rangle \\
\\
\langle factor \rangle & ::= & (\ \langle expr \rangle\ ) \mid \langle idf \rangle \\
\\
\langle idf \rangle & ::= & \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}
\end{array}
$$

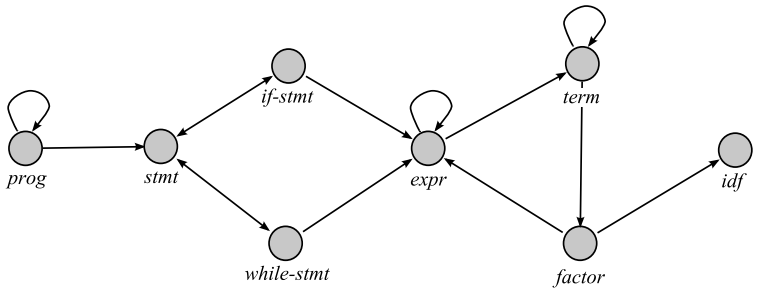


Figure 4.7: Example of a grammar and its derivability graph

1. Restrict the set of non-terminal symbols V_N to those non-terminals that belong to C.

2. Choose some start symbol $S_C \in C$ (usually the non-terminal most frequently referred from outside the component).

3. Restrict the set of production rules R to R_C containing only rules of a form $A \to \alpha$ where $A \in C$.

4. Transform all non-terminals $A \notin C$ that appear in the body of some rule from R_C into terminal symbols.

The point 4. is the most important. It says that all non-terminals outside the component are treated as terminals in the production rules of the induced grammar. We denote these special terminals as $\mathbf{t}_A$ in such a way that $\mathbf{t}_A$ corresponds to non-terminal A.

Lemma 4.4. *Grammar induced by a derivability component C is consistent if it assigns zero probability to infinite derivations and if for every $\mathbf{t}_A$ derivable from S_C grammar induced by comp(A) is consistent.*

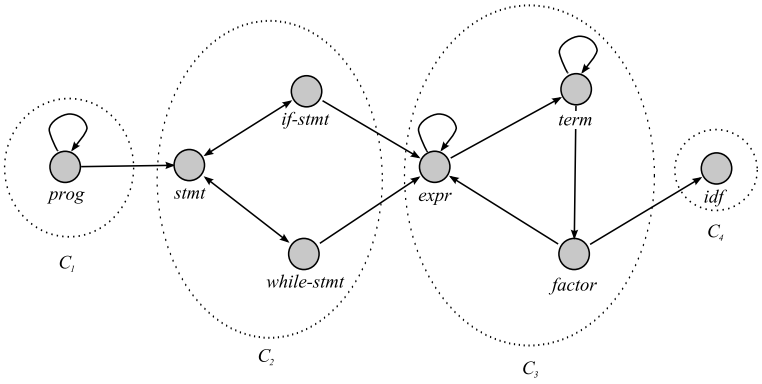
We can form a new graph by condensing each derivability component into a single vertex and by adjusting edges accordingly[4]. The resulting graph will be a directed acyclic graph (DAG) (see Figure 4.8). For every DAG there is at least one topological ordering of its vertices [12]. We will consider these orderings as topological orderings of derivability components.

Definition 4.14. We say that $C_1, C_2, \cdots, C_m$ is a *topological ordering* of derivability components of grammar G if for every i, j, $1 \leq i < j \leq m$ there exist two non-terminals $A, B \in V_N$ such that $A \in C_i$, $B \in C_j$ and there is a directed path from A to B in the derivability graph of G.

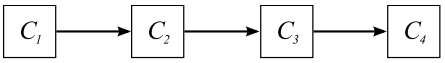
If we assume that the non-terminal symbols of the grammar G are grouped by derivability components to which they belong and that derivability components are ordered topologically as $C_1, C_2, \cdots, C_m$, the stochastic expectation matrix has a block triangular form

$$\mathbf{E} = \begin{bmatrix} \mathbf{E_1} & * & \cdots & * \\ 0 & \mathbf{E_2} & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{E_m} \end{bmatrix} \tag{4.5}$$

[4] We put a directed edge from C_i to C_j if there are some vertices $v_A \in C_i$ and $v_B \in C_j$ connected by an edge from v_A to v_B in the original derivability graph.



(a) original derivability graph



(b) graph of derivability components

Figure 4.8: Derivability components of the grammar from Figure 4.7

where $\mathbf{E_1}, \mathbf{E_2}, \ldots, \mathbf{E_m}$ are stochastic expectation matrices for grammars induced by derivability components $C_1, C_2, \cdots, C_m$ respectively.

Note. All elements of the block matrix on the right-hand side of 4.5 are matrices. Zeros ($\mathbf{0}$) stand for zero matrices of appropriate dimensions, each $\mathbf{E_i}$ is a $n_i \times n_i$ square matrix and star symbols ($*$) stand for non-zero matrices that we are not interested in (anonymous matrices).

Matrix $\mathbf{E}$ has zeros below the main block diagonal because derivability components are ordered topologically. From a given non-terminal we can derive only non-terminals belonging to derivability components that come later in the topological ordering. For all earlier non-terminals the expected number of occurrences is necessarily 0. Matrices $\mathbf{E_1}, \mathbf{E_2}, \ldots, \mathbf{E_m}$ on the main diagonal of $\mathbf{E}$ are by definition stochastic expectation matrices of individual derivability components.

Lemma 4.5. *Assume* $\mathbf{A}, \mathbf{B}, \mathbf{C}$ *are matrices of dimensions* $k \times k, k \times l, l \times l$ *respectively. Then*

$$\det(\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ 0 & \mathbf{C} \end{bmatrix}) = \det(\mathbf{A}).\det(\mathbf{B})$$

Corollary 4.6. *Determinant of a block triangular matrix is equal to the product of determinants of blocks on the main diagonal.*

In the previous section we have shown that grammar G is consistent if and only if $\rho(\mathbf{E}) < 1$. By definition of spectral radius

$$\rho(\mathbf{E}) = \max_{k=0,\ldots,n} |\lambda_k|$$

where $\lambda_1, \lambda_2, \ldots, \lambda_n$ are eigenvalues of the matrix $\mathbf{E}$. Eigenvalues of $\mathbf{E}$ are exactly the roots of the polynomial function $\det(\lambda\mathbf{I} - \mathbf{E}) = 0$. That is equivalent to

$$\det(\begin{bmatrix} \lambda\mathbf{I} - \mathbf{E_1} & * & \cdots & * \\ 0 & \lambda\mathbf{I} - \mathbf{E_2} & \cdots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda\mathbf{I} - \mathbf{E_m} \end{bmatrix}) = 0$$

and according to the lemma

$$\prod_{i=0}^{m} \det(\lambda\mathbf{I} - E_i) = 0$$

Therefore eigenvalues of $\mathbf{E}$ are the roots of equations $\det(\lambda\mathbf{I} - \mathbf{E_i}) = 0$ for $i = 0, \ldots, m$. That is exactly the union of the eigenvalues of matrices $\mathbf{E_1}, \mathbf{E_2}, \ldots, \mathbf{E_m}$. It follows immediately that

$$\rho(\mathbf{E}) < 1 \Leftrightarrow \forall i = 0, \ldots, m : \rho(\mathbf{E_i}) < 1$$

and as a consequence grammar G is consistent only if all of its derivability components $C_1, C_2, \ldots, C_m$ are consistent.

4.5 Context-sensitive Generation

Up to now, we have described sentence generation methods focusing on context-free aspects of programming languages. Strings of symbols generated using these methods are guaranteed to belong to the context-free superset of target programming language as defined in Section 2.2.1, but not necessarily to the language itself. Here we turn our attention to source code generation methods that take both context-free and context-sensitive syntax rules into consideration. Most of the presented ideas come from the prototype implementation of the Pascal source code generator that we call PASGEN – see Section 5.2. In this section we attempt to generalize all the concepts, methods and techniques to make them applicable independently of the target programming language.

4.5.1 Architecture of Context-sensitive Generator

To some extent, the source code generation process can be seen as an inverse of the source code analysis performed by compiler front-ends. Source code analysis is traditionally divided into three phases – lexical analysis, syntactic analysis and semantic analysis. The compiler front-end goes through these phases to gain full "understanding" of the given source code. This is necessary to preserve the original meaning of the input program when it is translated to machine code[5]. We can say that the product of a compiler front-end is a full semantic information about the source program that is subsequently consumed by a compiler back-end. The semantic information is usually conveyed in a form of an annotated abstract syntax tree (AST) that we call *semantic skeleton*.

To provide a clean separation of responsibilities we propose to structure the context-sensitive source code generators into two parts – the *AST generator* and the *back-end*. The AST generator is responsible for producing a semantic skeleton, i.e. an annotated AST. The back-end transforms the semantic skeleton into a concrete textual representation of the source program. The transformation consists of three phases which correspond with the inverted structure of a compiler front-end. The three phases are

1. Inverse semantic phase
2. Inverse syntactic phase
3. Inverse lexical phase

Each phase introduces some degree of variability. In the semantic phase the source code generator transforms the abstract syntax tree (semantic skeleton) into a concrete syntax tree. Programming language rules like operator precedence, variable naming and others are applied here. A common responsibility of the semantic phase is to generate parenthesis in expressions according to the operator precedence rules for the given programming language. The generator is also allowed to add redundant parenthesis and other constructs in places where they do not influence the semantics. The syntactic phase transforms the concrete syntax tree into a sequence of tokens according to the syntactic rules of the language. The resulting sequence of tokens is finally transformed into a textual representation in the lexical phase. Lexical phase is allowed to choose the concrete representation of tokens as well

[5] In our analogy, we do not consider compilers that translate from one high-level programming language to another. The vast majority of compilers are targeted to produce low-level machine code.

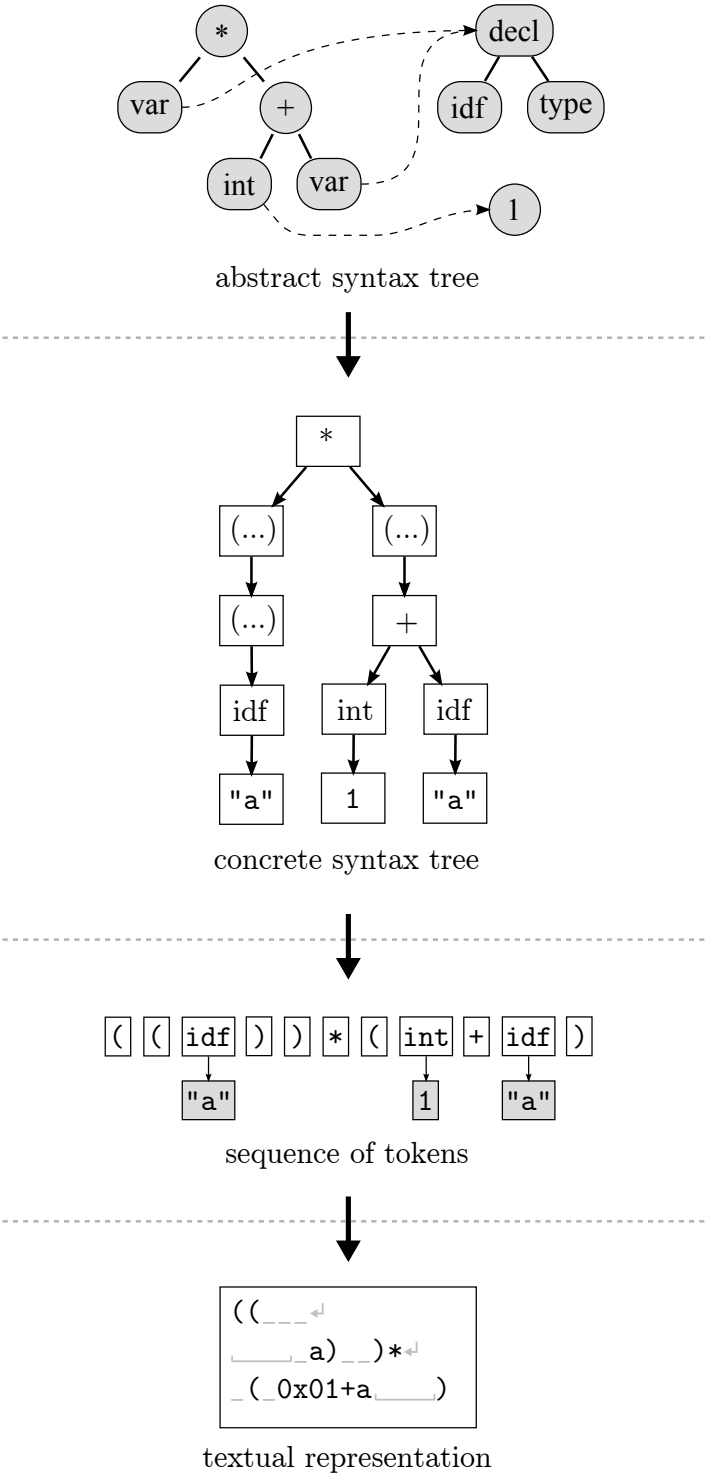


Figure 4.9: Transformation of AST into source code in three phases

as insert arbitrary whitespace between them. For example, a token representing integer constant 1 can be written as “1”, “01”, ”1L”, ”0x01”, etc. in different programming languages. The gradual transformation from AST to final textual representation is shown in Figure 4.9.

Back-end transformation phases are usually implemented by a simple traversal of the source representation, i.e. traversal of a syntax tree or iteration through a sequence of tokens. In the rest of this chapter we will focus on AST generation and related problems.

4.5.2 AST Generation Methods

4.5.2.1 Recursive Descent Generation

Recursive descent generation is a programming technique inspired by the recursive descent parsing technique. A generator using this technique is structured into procedures that correspond to the non-terminal symbols of the underlying context-free grammar G. For every non-terminal $A \in V_N$ there is a procedure gen_A responsible for generating sentences derivable from A. When a procedure gen_A is called, it returns a representation of one specific sentence α such that $A \overset{*}{\Rightarrow} \alpha$. To accomplish its task, gen_A recursively calls other procedures (and potentially also calls itself) to generate parts of the sentence α.

The recursive descent generation technique can be also used for AST generation. In this case generative procedures return specific AST fragments which are used to build parts of the AST. For example a generative procedure for expressions returns an AST subtree corresponding to one expression.

The main contribution of the recursive descent generation technique is that it structures the implementation of the generator along the natural boundaries of the problem. In case of the context-sensitive AST generation, we usually deal with different context-sensitive rules for each syntactic construct. Therefore dividing the implementation logic according to the underlying context-free grammar becomes very practical. Also since the problem of source code generation is intrinsically recursive this technique simplifies the construction of the generator by leveraging the support for recursive procedures in the implementation language.

On the other hand, applying recursive descent generation technique makes the generator heavily dependent on a concrete context-free grammar of the target programming language. Hence the applicability of this technique is limited to language specific source code generators.

Note. This technique can be applied only when implementing the source code generator in a programming language that supports procedures and

procedural recursion. Nearly all current programming languages satisfy this condition, however.

4.5.2.2 Left-to-right AST Generation

The left-to-right AST generation method defines the order, in which the nodes of the AST are generated. The motivation is to generate AST representations of the programming language constructs (e.g. declarations, statements, expressions) in the same order as they appear in the source code. We assume that the context for generating a subtree of the AST, i.e. a new construct, is always given by the previous parts of the AST that are already fully generated and therefore fixed. In other words, the AST nodes are generated in the same order as is produced by a depth-first traversal of the AST where subtrees of each node are traversed from left to right.

Example In Pascal, variables have to be declared before their use. Declarations that are in effect inside a given block are placed in front of this block or in front of some block that wraps this block. Therefore at the point in the program when the left-to-right AST generator is about to generate a variable reference, all effective variable declarations are already fixed. The generator is not allowed to create new declarations so it has to produce a variable reference corresponding with one of the existing declarations.

However, the left-to-right AST generation method makes essentially the same assumptions as one-pass compilers. Therefore this method is applicable only if the target programming language can be compiled by a one-pass compiler. Some programming languages have been designed specifically to be compiled by one-pass compilers, and include special constructs to allow one-pass compilation. An example of such a construct is the `forward` declaration in Pascal. Unfortunately, more advanced modern programming languages usually cannot be compiled in a single pass, as a result of their design.

4.5.3 Inverse Symbol Tables

Symbol tables are a data structure traditionally used by compilers to hold information about entities that appear in the source program such as their identifiers, types, position in the storage and other relevant information [1]. These information are collected incrementally during the analysis phases of the compiler. The most common operation on standard symbol tables is to retrieve these information for an identifier or to check whether an entity with a given identifier is already defined in the current scope.

In case of a compiler, the identifier is known in advance and the symbol table is used to retrieve information about the entity denoted by this identifier. These information are used to check whether the context-sensitive rules are satisfied, i.e., whether it is allowed for the denoted entity to appear at the given place in the program (context). However, during a context-sensitive source code generation the requirements are reversed. Instead of checking whether the rules are satisfied for a concrete occurrence of an identifier, the generator needs to find the set of identifiers that are allowed to appear in the given place and choose one of them to be generated. We call the data structure that supports these operations in a source code generator *inverse symbol table*.

Inverse symbol table is essentially a register of declared entities searchable according to different criteria, particularly the criteria important for the context-sensitive rules (e.g. scoping rules, typing rules,...). A possible symbol table query might be to list all entities of a given type visible in the given scope according to the current scope nesting hierarchy.

4.5.4 Typing Rules

If the target programming language uses a static type system, then typing rules usually pose the most difficult problem for context-sensitive source code generation. In order to generate well-typed programs, the generator has to keep track of types during the program generation.

4.5.4.1 Type Constraints

Type constraints are a method that can be used in recursive descent left-to-right AST generation to deal with the static typing rules for expressions. The assumption is that the set of types that an expression is allowed to have is determined by the current context when the generator decides to create it. For example, a subexpression that stands as an operand of a logical conjunction operator is expected to have a Boolean type. *Type constraint* is an explicit representation of the set of types that are allowed in the given context. Typical type constraints include:

- T has to be a numeric type
- T has to be an atomic type (as opposed to composite type)
- T has to be type-compatible with the built-in Boolean type

Type constraints are used in a recursive descent generator to influence the choice of production rules in generative procedures. The generator performs

the rule selection step with respect to the current effective type constraint. Depending on the context, only the rules matching the type constraint are allowed to be selected. A production rule matches a type constraint if its body can be potentially expanded into a programming language construct (expression) that has a proper type, i.e. one of the types represented by the type constraint. After a matching production rule is chosen, the generator prepares a new type constraint to be used for every non-terminal in the body of the rule. Note that different type constraints can be used for different non-terminals and they might or might not be related with the current effective type constraint.

In order to influence the choice of production rules, the current effective type constraint has to be accessible to the rule selection mechanisms of the generator. It can be passed as an argument to the respective generative procedures or be a part of the global generator environment.

Example Consider the conditional expression construct, which appears in many programming languages. It usually consists of three expressions where the first expression is called the "condition" and the other two are called "alternatives". If the condition evaluates to *true*, it yields the value of the first alternative, or else, if the condition evaluates to $false$, it yields the value of the second alternative. A common syntax for conditional expressions looks like this

$$\langle \textit{conditional-expression} \rangle ::= \langle \textit{expression}_1 \rangle \ \mathbf{?} \ \langle \textit{expression}_2 \rangle \ \mathbf{:} \ \langle \textit{expression}_3 \rangle$$

The typing rules for this construct state that it is well-typed only if the condition has a Boolean type and the type of both alternatives is the same (or they have a common supertype). The type of the conditional expression itself is the same as the type of both alternatives (or as their least common supertype). According to these typing rules, when a generator using the type constraint method wants to produce a conditional statement, it has to propagate the current type constraint to both alternatives. However, for the expression representing the condition, a completely independent type constraint representing the Boolean type has to be used. Therefore in order to generate a conditional expression, the same generative procedure for expressions is called three times – each time with a different type constraints.

It is important to understand precisely what does it mean that a production rule matches a given type constraint. The set of types that can be potentially obtained by expanding the body of the rule in fact consists of *all* types allowed by the static typing rules for the given programming language constructs. It is defined statically regardless of the concrete context. Therefore it is a superset of all types which can be actually obtained in the current

context. For example, the function call construct can potentially produce expression of any possible type. However, the set of types which can be actually obtained by a function call is restricted at least by the set of function definitions visible in the current context.

Problems In specific contexts it is possible for the generator to get into an *impossible choice* situation, i.e. a situation where all the available choices are ruled out by the type constraints. This happens when previous choices cause the generator to get into a context where it needs to generate a construct of some specific type, but no such construct can be generated because of the context-sensitive rules.

The following example illustrates this problem on a hypothetical Pascal source code generator (see also Section 5.2). Assume that the generator has chosen to produce a Pascal function calls construct referring to a function f that has one argument of type T. In this context the generator needs to produce an expression of type T that would be used as an argument in function call construct. However, if there is no variable of type T declared and if there is no other way to produce an expression of type T, the generator is in an impossible choice situation.

```
program ImpossibleChoise;

type T = record
  x,y : Integer;
end;

function f(arg1 : T) : Integer;
  begin
    f := 0;
  end;

begin
  f( ??? )
end.
```

The three question marks in the sample program point out the location where the impossible choice occurs. This is because there is no way to generate proper argument for the function call to f.

Even less desirable than the impossible choice situation is the related *pathological choice* situation. The pathological choice situation occurs when

the type constraints do not exclude all production rule choices, but the remaining rules can neither lead to finite termination of the generation process nor satisfy the current type constraints. How is the occurrence of such a situation possible? The reason is that the generator is allowed to select a rule which can satisfy the current effective type constraint *potentially* (according to the static typing rules) but not *actually* (according to the current context).

Unfortunately, these problems can not be easily avoided since they are inherent to recursive descent generation technique on which the type constraints method depends[6]. It is not directly possible for a recursive descent generator to find out that it is in a "pathological choice" situation or to prevent entering into a "impossible choice" situation. The generator has no easy way to determine whether it is actually possible to produce an expression of a given type in the current context. For illustration, in Pascal, the given type might be obtained by a function call, by a record field access, by an array element access, by a pointer dereference, or, most importantly, by any finite combination there of. To perform such a complicated check in a generative procedure for a specific construct (e.g. function call) would go against the design principle of being concerned only with the "local" context-sensitive rules. As discussed in Subsection 4.5.2.1, this principle forms the basis of the recursive descent generation technique. Therefore we have developed a new method for dealing with typing rules.

4.5.4.2 Combinators

To solve the problems of "impossible choice" and "pathological choice" that are related to the type constraints method and recursive descent generation technique, we develop a new method that helps with generation of well-typed programs. The method uses an abstraction called *combinators* to model the relationships between types in the current context. This allows the generator to efficiently compute the set of types which can be actually obtained by producing an expression in the current context [7]. We call these types *reachable types*.

Combinators are abstractions for constructs that constitute expressions in the given programming language. In the most general view, each combinator

[6] Recursive descent generator can recover from both "impossible choice" and "pathological choice" situations by employing limited backtracking and special heuristics. It's not an elegant solution, but it works in practice as demonstrated by the prototype implementation of a source code generator for Pascal – see Section 5.2.

[7] As opposed to the set of types which can be potentially obtained according to the typing rules.

is defined by a set of *required types* and a set of *provided types* (also referred to as parameter types and return types respectively). In practice the set of provided types usually contains exactly one type which we will call the *return type* of the combinator.

Definition 4.15. *Combinator* C is an ordered pair of sets $(\mathcal{T}_R, \mathcal{T}_P)$ where $\mathcal{T}_R$ is the set of required types and $\mathcal{T}_P$ is the set of provided types.

Combinator $C = (\mathcal{T}_R, \mathcal{T}_P)$ where $\mathcal{T}_R = \{T_1, T_2, \ldots, T_k\}$ and $\mathcal{T}_P = \{T_0\}$ is an abstraction for the expression construct that combines subexpressions of types $T_1, T_2, \ldots, T_k$ and produces an expression of type T_0. To motivate our terminology we can say that this construct "requires" expressions of types $T_1, T_2, \ldots, T_k$ as parameters and "provides" an expression of type T_0 in return.

Example Assume that the following Pascal declaration are in effect.

```
type TColor = (red, green, blue);
type TRange = 0..255;
type TArray = Array [TColor] of TRange;

var index : TColor;
var rgb : TArray;
```

To generate an expression of type `TRange`, it is necessary to access the array variable `rgb`. According to the definition of type `TArray`, the array access construct requires a subexpression of type `TColor` to be used as an index. An easy way to obtain such an expression is to generate a reference to the variable `index`.

```
var temp : TRange;
...
temp := rgb[index];
```

The given example can be described by the following three combinators.

$$\begin{aligned} C_1 &= (\emptyset, \{\texttt{TArray}\}) \\ C_2 &= (\emptyset, \{\texttt{TColor}\}) \\ C_3 &= (\{\texttt{TArray}, \texttt{TColor}\}, \{\texttt{TRange}\}) \end{aligned}$$

Combinator C_1 represents the possibility of generating an expression of type `TArray` by producing a reference to variable `rgb`. From the typing perspective this means that we can get the type `TArray` for free, i.e. without

requiring any other type. Similarly C_2 provides type `TColor` by referencing variable `index`. Combinator C_3 demonstrates the main point of combinators. It represents the possibility of generating an expression of type `TRange` by combining subexpressions of types `TArray` and `TColor`. The example shows how combinators abstract away the syntax of the programming language constructs and express only the relationships between types.

There are two ways how combinators can be utilized in the source code generator:

1. The set of reachable types for a given construct can be computed and incrementally updated by using combinators to model the current context. This allows to implement the rule selection procedures of the source code generator in such a way that impossible choice and pathological choice situations are prevented.

2. A new *type-driven method* for generating expressions can be adopted. The point of this method is to choose the final type of the expression from the set of reachable types *before* the expression is generated. When the final type T of the expression is given in advance, combinators can be used to drive the generation process. Since T belongs to the set of reachable types, it must be obtainable as some combination of the combinators. Formally, there must be a sequence of combinators $C_1, C_2, \ldots, C_n$ where $C_i = (\mathcal{T}_R^i, \mathcal{T}_P^i)$ such that $T \in \bigcup_{i=1}^{n} \mathcal{T}_P^i$ and

$$\forall i \in \{1, \ldots, n\} : T_R^i \subset \bigcup_{k=1}^{i-1} T_P^k$$

   This sequence can be seen as a recipe for the generator describing how to construct an expression of type T. In other words, an expression construct can be generated only if all required types of the corresponding combinator are reachable. Therefore the set of reachable types determines which constructs can be produced.

In an AST generator, a combinator $C = (\{T_1, T_2, \ldots, T_n\}, \{T\})$ can be seen as a routine that takes AST subtrees $t_1, t_2, \ldots, t_n$, which represent expressions of types $T_1, T_2, \ldots, T_n$, and produces an AST fragment, which represents an expression of type T. Combinators are utilized this way in the Pascal source code generator described in Section 5.2.

In the following paragraphs we briefly describe some of the most common combinators.

Variable Access Combinator The simplest combinator corresponds to the expression construct that references entire variable visible in the current scope. For a variable of type T it has the form

$$C = (\emptyset, \{T\})$$

Combinators of this kind always act as starting points since they do not have any required types. In other words, they provide type T for free.

Pointer Dereference Combinator This combinator corresponds to the pointer dereference construct. Pointer dereference requires a subexpression of some pointer type T_{ptr}. If $T_{ptr} = Ref\ T$, i.e., if T_{ptr} points to some type T, then the pointer dereference combinator has the form

$$C = (\{T_{ptr}\}, \{T\})$$

Field Access Combinator This combinator corresponds with the construct that allows accessing fields of record variables. It requires a subexpression of a record type T_{rec}. If the fields of the record type T_{rec} have types $T_1, \ldots, T_n$, then the field access combinator has one of the forms

$$C = (\{T_{rec}\}, \{T_i\})$$

where $i = 1, \ldots, n$.

Array Access Combinator This combinator corresponds with the construct that allows accessing components of array variables. It requires a subexpression of an array type T_{array} and a subexpression of type T_{idx} . If T_{idx} is the index type of T_{array} and the components of T_{array} have type T then the array access combinator has the form

$$C = (\{T_{array}, T_{idx}\}, \{T\})$$

Note. This variant of the combinators is applicable for programming languages that allow indexing of arrays with ordinal types other than integers. However, this is not allowed in most modern programming languages where T_{idx} always corresponds to the built-in integer type.

Function Call Combinator This combinator corresponds with the function call construct. For a function f that has arguments of types $T_1, T_2, \ldots, T_n$ and a return value of type T, the function call combinator has the form

$$C = (\{T_1, T_2, \ldots, T_n\}, \{T\})$$

Chapter 5

Prototype Implementations

Three prototype tools were created as part of this work to test practical applicability of the concepts and methods developed in the previous chapters. Two of these tools are source code generators while the third is a grammar analyzer. The following table gives an overview of the basic methods implemented in each of the tools.

	GRAMGEN	PASGEN	YAGA
symbol selection methods:			
depth-first (DFSS)		X	
breadth-first (BFSS)	X		
rule selection methods:			
random (RRS)	X		
probabilistic (PRS)	X	X	
finalization (FRS)	X		
analysis methods:			
SCFG consistency			X

Table 5.1: Feature overview of the prototype tools

GRAMGEN is a universal sentence generation tool driven by context-free grammars, PASGEN is a Pascal source code generator and YAGA is a tool for analyzing consistency of stochastic context-free grammars.

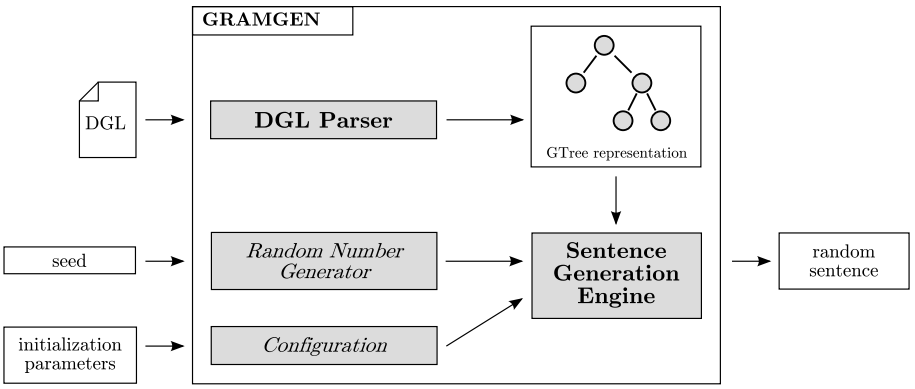


Figure 5.1: High-level overview of GRAMGEN run-time architecture

5.1 Grammar-driven Generator (GRAMGEN)

5.1.1 History

GRAMGEN was implemented in 2004 as a small research project sponsored by Lynguent, Inc[1]. At that time Lynguent focused on developing compilers and program transformation tools for hardware description languages (HDLs). GRAMGEN was created for testing front-end of a VHDL-AMS compiler. The front-end used a Bison-generated parser that was based on a heavily modified VHDL-AMS grammar. The modification were necessary in order to have the grammar in a LALR(1) form required by Bison. It is difficult and therefore highly error-prone to perform such modifications without changing the language described by the grammar. Since it is an undecidable problem to test whether two context-free grammars describe the same language, it was not possible to algorithmically verify that the modifications performed on the original VHDL-AMS grammar are correct. As an alternative testing with GRAMGEN was planned to be used to gain confidence that the parser really accepts the language defined by the VHDL-AMS language reference manual.

5.1.2 Architecture

GRAMGEN is a tool that generates random strings of symbols according to a given context-free grammar. The grammar is specified in a format called DGL which is quite similar to BNF notation. Before the sentence generation is started, GRAMGEN parses the DGL description and builds an in-

[1]See http://www.lynguent.com/.

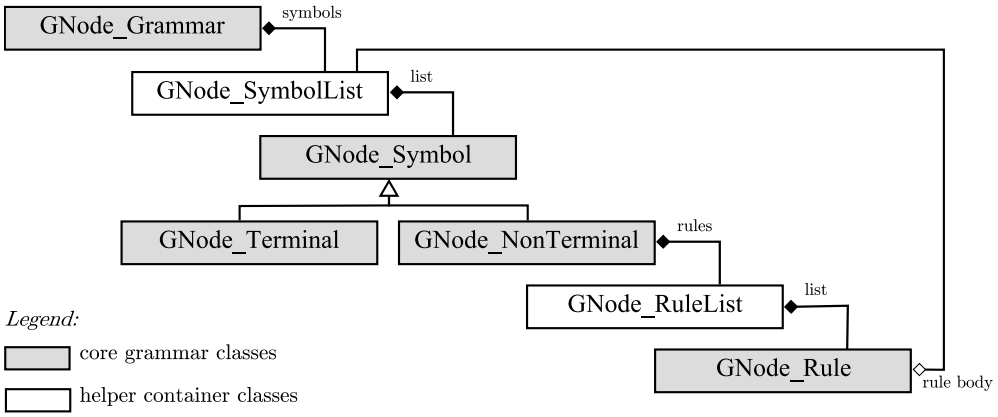


Figure 5.2: Representation of context-free grammars in GRAMGEN

memory object-oriented representation of the grammar. Classes constituting the grammar object model are commonly referred to as `GTree` data structure (see Figure 5.2). To perform randomized choices, GRAMGEN uses a pseudo-random number generator that can be initialized by a user-defined seed number. This allows for reproducible results.

Conceptually GRAMGEN is a direct realization of the breadth-first symbol selection method described in Subsection 4.2.2. The sentence generation algorithm works iteratively with two symbol buffers, SRC and `DST`. Before the first iteration SRC buffer contains just the start symbol of the grammar and DST buffer is empty. In each iteration SRC buffer is transformed into DST buffer. Each symbols from the SRC buffer is processed separately. Terminal symbols are moved directly to the DST buffer. For each non-terminal symbol a matching production rule is selected and the symbol is rewritten according to this rule. Rewriting the symbol means that the symbol is popped from the SRC buffer and the right-hand side of the selected production rule is pushed to the DST buffer. At the end of the iteration DST buffer becomes the new SRC and a new DST is created empty. The algorithm terminates when there are no more non-terminals is the DST buffer at the end of the iteration. When this happens, DST buffer contains a sequence of terminal symbols which constitute a terminal string belonging to the language described by the input grammar. GRAMGEN emits this resulting strings as output.

5.1.2.1 Rule Selection

Since there is no variable element in the breadth-first symbol selection method, the resulting sentence is fully determined by the production rule choices made by GRAMGEN's sentence generation engine. Engine chooses production rules

Algorithm 5.1 Scheme of the GRAMGEN sentence generation algorithm

```
while (...) do
  for each symbol A in SRC do

    if (A is terminal) then
      push A into DST
    endif

    if (A is non-terminal) then
      select rule R = (A -> w)
      push w into DST
    endif
  end

  SRC = DST
  DST = empty
end
```

according to the actual rule selection mode. Three different rule selection modes are available in GRAMGEN:

Random selection Generator selects a random production rule for any given non-terminal (among the production rules pertaining to the given non-terminal). This is the default rule selection mode. Corresponds to `GNode_RuleList::SELECT_RANDOM`.

Weighted selection Generator selects a random production rule with respect to the given probability distribution. The probability distribution is given by the weights of individual production rules. Underlying concepts are described in Section 4.3.2. This mode corresponds to `GNode_RuleList::SELECT_WEIGHT`.

Finalization selection Generator selects one of the *finalization rules* for any given non-terminal. Finalization rules are the rules that lead to shortest possible terminal string for the given non-terminal. More about finalization rules can be found in Subsection 4.3.3. In order to use the finalization selection mode finalization rules have to be precomputed before the generation process is started. This mode corresponds to `GNode_RuleList::SELECT_FINALIZE`.

```
var: 1 : "a", 1 : "b", 1 : "c";

expr:   20 : "(" %{expr} ")",
        2  : "-" %{expr},
        10 : %{expr} "+" %{expr},
        10 : %{expr} "-" %{expr},
        10 : %{expr} "*" %{expr},
        10 : %{expr} "/" %{expr},
        60 : %{var};
```

Figure 5.3: Sample DGL grammar for arithmetic expressions

The rule selection mode can be dynamically changed during the sentence generation. This way selection modes can complement each other. "Use cases" in GRAMGEN:

- Random or weighted selection mode is used for a given fixed number of iterations specified by the user in initialization parameters. After the given number of iterations the generation engine is switched into finalization selection mode to complete the sentence in smallest possible number of iterations.
- Random or weighted selection mode is used until the size of the DST buffer reaches the user specified limit (or until the generator arrives at a terminal sentence). When the given "soft limit" on the length of the sentence is reached the generation engine switches to finalization selection mode to complete the sentence in smallest possible number of iterations. With a little bit of additional work a "hard limit" on length of the produced sentence could be also enforced.

5.1.2.2 Data Generation Language (DGL)

Modified subset of *data generation language (DGL)* was chosen in GRAMGEN as a format for specifying the context-free grammar. DGL was developed by Peter M. Maurer for general functional level testing of VLSI designs [29, 30, 31]. He also developed a data generator based on DGL language and released the sources to the public domain. Although Maurer's generator has a wide range of features, it is not appropriate for our main goal, i.e. to generate source code of high-level programing languages such as Pascal, C, VHDL etc. GRAMGEN was implemented without any use of Maurer's source code.

$$
\begin{aligned}
\langle \textit{production} \rangle &::= \langle \textit{non-terminal} \rangle \ \mathbf{:} \ \langle \textit{rules} \rangle \ \mathbf{;} \\
\langle \textit{rules} \rangle &::= \langle \textit{rule} \rangle \mid \langle \textit{rules} \rangle \ \mathbf{,} \ \langle \textit{rule} \rangle \\
\langle \textit{rule} \rangle &::= \langle \textit{weight} \rangle \ \mathbf{:} \ \langle \textit{symbols} \rangle \\
\langle \textit{symbols} \rangle &::= \langle \textit{symbol} \rangle \mid \langle \textit{symbols} \rangle \, \langle \textit{symbol} \rangle \\
\langle \textit{symbol} \rangle &::= \langle \textit{string} \rangle \mid \mathbf{\%\{} \ \langle \textit{non-terminal} \rangle \ \mathbf{\}} \mid \langle \textit{symbol-ext} \rangle \\
\langle \textit{string} \rangle &::= \text{”} \ \langle \text{sequence of characters except " and newline} \rangle \ \text{”} \\
\langle \textit{non-terminal} \rangle &::= \langle \text{sequence of characters a–z, A–Z, 0–9} \rangle \\
\langle \textit{weight} \rangle &::= \langle \text{non-negative integer number} \rangle \\
\\
\langle \textit{symbol-ext} \rangle &::= [\ \langle \textit{symbols} \rangle \ ] \mid \mathbf{\{} \ \langle \textit{symbols} \rangle \ \mathbf{\}}
\end{aligned}
$$

Figure 5.4: Syntax of DGL productions

GRAMGEN supports a limited subset of DGL with minor modification as defined by Figure 5.4. This subset is roughly equivalent to the BNF notation with the addition of production rules weights to be able to express WCFG (see Definition 4.2). Input grammar written in DGL is a sequence of productions. Each ⟨*production*⟩ groups together ⟨*rules*⟩ for one ⟨*non-terminal*⟩ symbol of the grammar. The first production corresponds with the start symbol. Each rules consists of a sequence of ⟨*symbols*⟩. It may be optionally preceded by a numeric ⟨*weight*⟩ specification. Terminal ⟨*symbol*⟩s are expressed as literal ⟨*string*⟩s enclosed in quotes. Non-terminal ⟨*symbol*⟩s are enclosed in braces and preceded with % character.

The presented subset of DGL was extended with two EBNF-like features which simplify specification of programming language grammars:

- Any part of the ⟨*rule*⟩ can be made *optional* by enclosing it in brackets (0..1 occurrences).
- Any part of the ⟨*rule*⟩ can be made *repetitive* by enclosing it in brackets (0..n occurrences).

Syntax of both extensions is defined by rule ⟨*symbol-ext*⟩ in Figure 5.4.

GRAMGEN is implemented in C++. Instructions how to build and run GRAMGEN can be found in Appendix A.

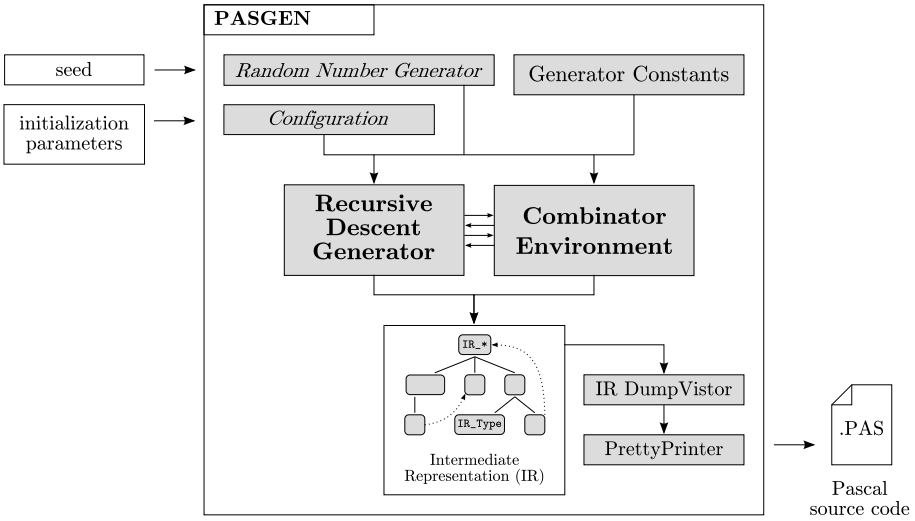


Figure 5.5: High-level overview of PASGEN run-time architecture

5.2 Pascal Source Code Generator (PASGEN)

PASGEN is a language-specific source code generator for Pascal that focuses on context-sensitive syntax rules. It attempts to generate syntactically correct programs that are compliant with the ISO/IEC 7185:1990 specification of the Pascal programming language [20]. The emphasis is on the context-sensitive syntax rules which govern the following Pascal constructs[2]:

1. Blocks (6.2.1) and scopes (6.2.2)
2. Constant definitions (6.3)
3. Type definitions (6.4) including forward pointer types (6.4.4)
4. Variable declarations and variable access (6.5)
5. Type compatibility (6.4.5) and assignment compatibility (6.4.6)

Generally, the implementation of PASGEN serves as a demonstration of the theoretical principles developed in Section 4.5.

[2]The numbers in the parenthesis identify the corresponding sections of the Pascal ISO/IEC 7185:1990 specification [20].

5.2.1 Architecture

In terms of context-free generator framework developed in Section 4.1 PASGEN is an instantiation of the abstract sentence generation algorithm using the depth-first symbol selection method and the probabilistic rule selection method. The rule selection step contains additional modifications to account for context-sensitive syntax rules.

More precisely PASGEN is primarily implemented as a recursive descent generator using the technique described in Subsection 4.5.2.1. The structure of the generator corresponds with a concrete context-free grammar of Pascal. For each non-terminal symbol of the grammar, there is a separate method which implements the associated production rules, i.e., the rules rewriting the given non-terminal symbol[3]. The choice of a production rule in each method is performed pseudo-randomly according to predefined probabilities. An example of a generative methods is shown in the following simplified code excerpt from the main class of PASGEN.

```
public generateStatement() {

    float prob_if    = 25; // if-statement
    float prob_while = 15; // while-statement
    float prob_assign = 40; // assignment-statement
    ...

    switch(choice(
           prob_if,
           prob_while,
           prob_assign
           ...
            )) {

    case 0: generateIfStatement(); break;
    case 1: generateWhileStatement(); break;
    case 2: generateAssignmentStatement(); break;
    ...
    default: throw new AssertionError();
    }
}
```

[3] As mentioned in Subsection 4.5.2.1 this implementation technique implies that PASGEN is bound to a fixed grammar.

In Subsection 4.5.1 we describe an architecture for context-sensitive source code generators. PASGEN follows this architecture in many aspects. The core recursive descent left-to-right generator produces an *intermediate representation* (IR) of the source code, which essentially an annotated AST. The IR implements the visitor pattern [14] and hence it can easily transformed into a concrete textual form by a so-called *dump visitor*. The dump visitor traverses the IR tree and performs some of the backend tasks defined in Subsection 4.5.1 including pretty-printing.

5.2.2 Intermediate Representation (IR)

In PASGEN, the intermediate representation presents an object-oriented model for Pascal syntax trees. Since PASGEN is implemented in Java, there is special Java class for each type of IR nodes. The following example shows how the IR tree is constructed.

```
IR_Type type = generateType();

IR_Identifier variableName = new IR_Identifier(Type.VARIABLE);
IR_VariableDeclaration variableDecl =
    new IR_VariableDeclaration(variableName, type);

symbolTable.insert(variableName, variableDecl);
```

An interesting feature of the IR is that it uses Java class inheritance to express hierarchical relationships between types in Pascal. Each type is represented by an object of class `IR_Type` or some of its descendants (see Figure 5.6). This is related to the implementation of type constraints is PASGEN – see Subsection 4.5.4.1. Sometimes we need to query for types of a specific kind (e.g. for all record types or for all pointer types). Such queries are implemented by simply returning types whose IR representation is an instance of the respective class (e.g. `IR_RecordType` or `IR_PointerType`). The inheritance hierarchy of `IR_Type` subclasses also allows us to form queries for broader kinds of types (e.g. all simple types). This is done by searching for types whose IR representation inherits form the appropriate abstract class (e.g. `IR_SimpleType`).

5.2.3 Symbol Tables

Symbol tables in PASGEN are different from conventional symbol tables used in compilers. The concept of *inverse symbol tables* is applied as described

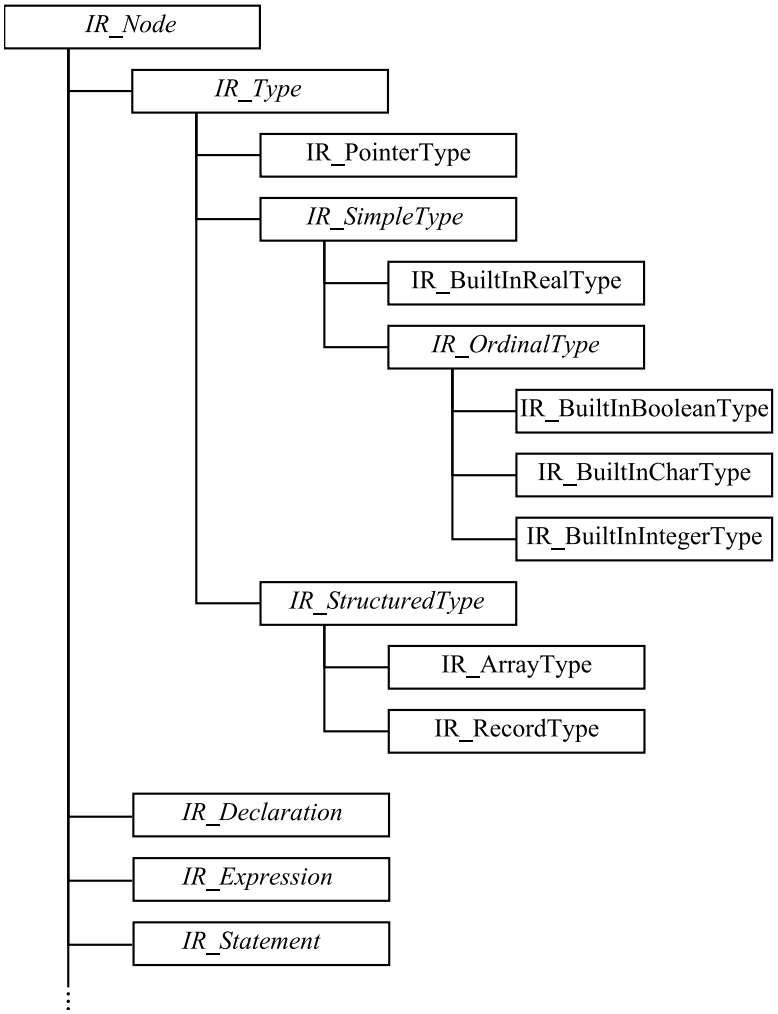


Figure 5.6: Class hierarchy of `IR_Type` subclasses

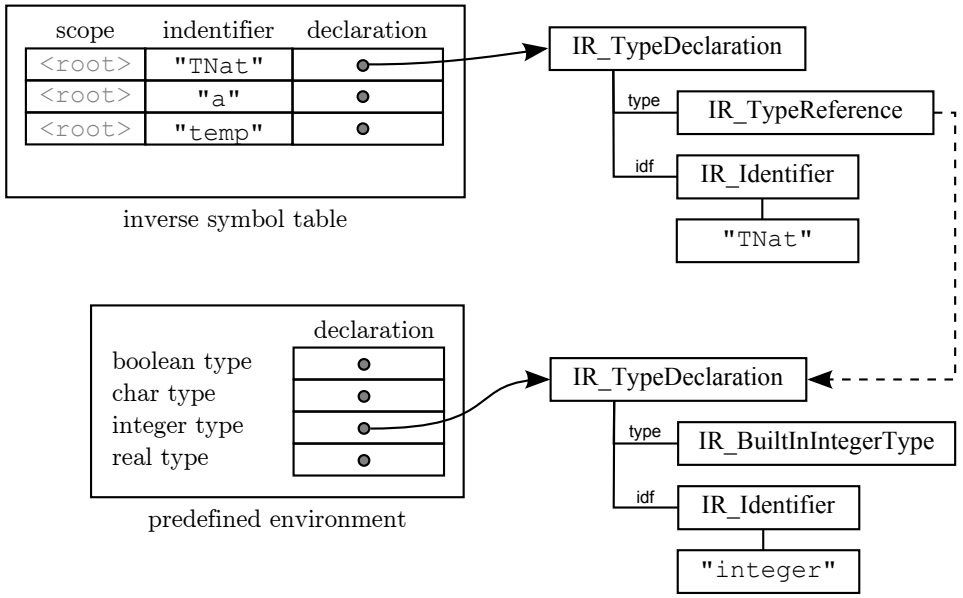


Figure 5.7: Inverse symbol tables and predefined environment in PASGEN

in Subsection 4.5.3. Conventional symbol table is a data structure used by compiler to hold information about an entity from the source program such as its identifier, its type, its position in the storage and any other relevant information [1]. The information is collected incrementally during the analysis phases of the compiler. While the inverse symbol tables in PASGEN hold similar information as shown in Figure 5.7, they are used in a different way.

When PASGEN wants to generate an entity reference (e.g. a variable access) is uses the inverse symbol tables to search for all entities that can be referenced at the given point in the program. The inverse symbol tables return a list of such entities from which PASGEN randomly chooses one entity for the entity reference construct.

5.2.4 Type Constraints and Combinators

To handle Pascal static typing rules PASGEN uses both the method of type constraints described in Subsection 4.5.4.1 and the combinators method described in Subsection 4.5.4.2. The type constraints were used as the initial solution that was later partially superseded by combinators. This was mainly because of the problems related to impossible choice and pathological choice situations as discussed in Subsection 4.5.4.1.

Technically, the type constraints are passed along the generative methods as parameters, while the combinators are stored as a part of the global gener-

Package	*Description*
`thesis.poc.pasgen`	Main classes
`thesis.poc.pasgen.tree.*`	Nodes of the intermediate representation tree
`thesis.poc.pasgen.comb`	Classes implementing combinators
`thesis.poc.pasgen.util`	Utility classes

Figure 5.8: Structure of packages in PASGEN implementation

ator environment. For more details about type constraints and combinators please refer to the relevant section in the previous chapter.

5.2.5 Implementation Notes

PASGEN is implemented in Java™. The structure of Java packages is shown in Figure 5.8. The core functionality does not rely on other 3rd party libraries. Utility libraries are listed in Appendix A.

5.3 Grammar Analyzer (YAGA)

The source code generation techniques developed in this work are all based on context-free grammars and their variants. As discussed in previous sections the recursive nature of context-free grammars makes the generation algorithms susceptible to divergence problems. To circumvent these problems we need to gain more control over the generation process. This is done by introducing production rule probabilities into the grammars turning them into stochastic context-free grammars (see Definition 4.3). It was shown in Section 4.3.2 that finite termination of the can be guaranteed when the grammar used for generation is a *consistent* stochastic context-free grammar. Consistency is therefore an important property of a grammar determining its suitability for source code generation.

YAGA (Yet Another Grammar Analyzer) is a prototype implementation of a tool for determining consistency of stochastic context-free grammars and various other metrics. It takes a formal description of a SCFG on input, performs analysis on the specified grammar and produces an analysis report as a result. The formal description of the SCFG can be given in two formats:

DGL grammar DGL has built-in syntax for specifying weights of individual production rules. Hence the grammar defined in DGL is essentially a weighted context-free grammar (WCFG) by Definition 4.2. YAGA normalizes the weighted context-free grammar into a stochastic context-

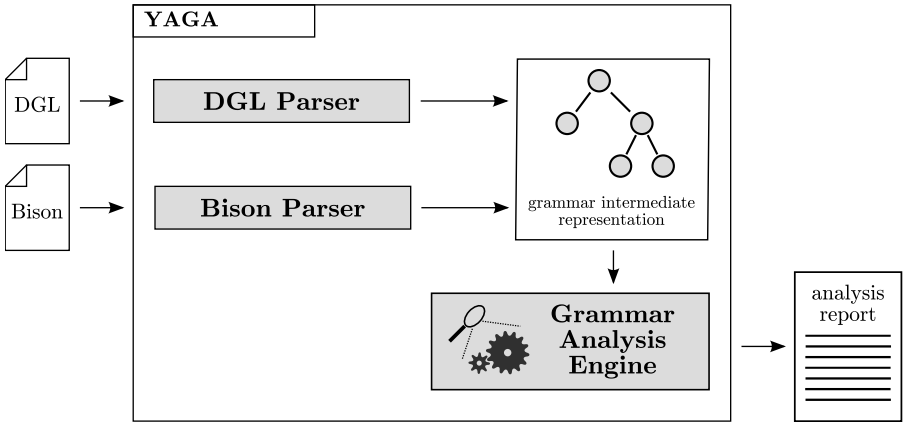


Figure 5.9: High-level overview of YAGA run-time architecture.

free grammar according to 4.1. DGL is also used by GRAMGEN as input format for grammar descriptions.

Bison grammar Bison is a well-known parser generator tool. It can automatically generate a C/C++ parser for any given LALR(1) grammar. The grammar is given in Bison-specific format which resembles the BNF notation. YAGA can parse the Bison grammar specification and perform analysis on the extracted grammar. There is a large number of grammars available, since virtually every modern parser implemented in C/C++ includes a Bison grammar. It very useful that YAGA is able to analyze these grammars directly without a need to transform them into some other specific format.

Besides checking the consistency of the given SCFG, the analysis report produced by YAGA contains the following information:

- Basic statistical information about the given grammar
- Decomposition of the grammar into derivability components – see Subsection 4.4.2
- Separate consistency verification for every derivability component (modular consistency)

5.3.1 Architecture

YAGA consists of three logical parts: grammar parser infrastructure, grammar model and grammar analysis engine. It has a loosely coupled architecture. The analysis engine operates on the intermediate representation

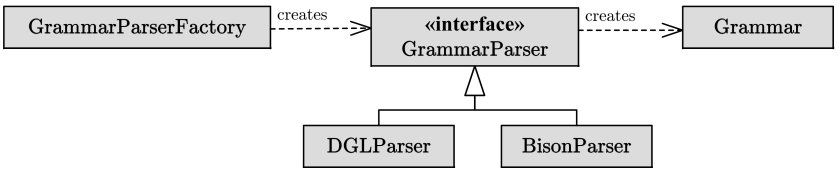


Figure 5.10: Parser infrastructure classes in YAGA

of the grammar and is therefore completely independent of the way how the intermediate representation is constructed. The usual way is to use the grammar parser infrastructure. The parser infrastructure provides a thin layer of abstraction over concrete parser implementations (see Figure 5.10). The `GrammarParserFactory` creates an appropriate implementation of the `GrammarParser` interface according to the specified `GrammarFormat`. The concrete parser implementation is subsequently accessed only through the simple generic `GrammarParser` interface.

The grammar object model in YAGA (see Figure 5.11) is similar to GTree grammar object model used in GRAMGEN. It has a cleaner design and implementation thanks to the lessons learned from GRAMGEN implementation. The grammar object model is not just an in-memory representation of the grammars. It provides functionality specifically tailored for the needs of the analysis engine. Besides that it is designed to allow easy programmatic creation of small prototype grammar model instances.

Example

```
Grammar grammar = new Grammar();

Terminal PLUS = new Terminal("'+'");
Terminal MINUS = new Terminal("'-'");
Terminal NUM = new Terminal("NUM");
Terminal VAR = new Terminal("VAR");

NonTerminal expr = new NonTerminal("expression");
expr.addProduction(new Production(0.15, expr, PLUS, expr));
expr.addProduction(new Production(0.15, expr, MINUS, expr));
expr.addProduction(new Production(0.30, NUM));
expr.addProduction(new Production(0.30, VAR));

grammar.addSymbols(expr, PLUS, MINUS, NUM, VAR);
grammar.setStartSymbol(expr);
```

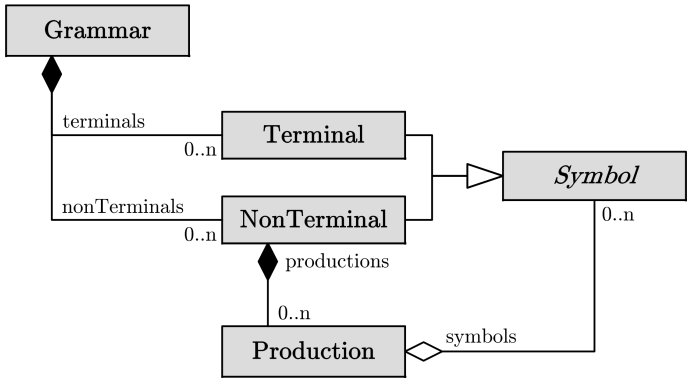


Figure 5.11: Object-oriented model of context-free grammars in YAGA

This Java code fragment creates a grammar object model equivalent to the following stochastic context-free grammar.

$$\begin{array}{rcll} \langle expression \rangle & ::= & (0.15) & \langle expression \rangle + \langle expression \rangle \\ & | & (0.15) & \langle expression \rangle - \langle expression \rangle \\ & | & (0.30) & \textbf{NUM} \\ & | & (0.30) & \textbf{VAR} \end{array}$$

The grammar analysis engine provides an implementation of the algorithms described in Subsection 4.4.1 and Subsection 4.4.2 The algorithms work directly the grammar object model.

5.3.2 Implementation Notes

YAGA is implemented in Java™. The structure of Java packages is shown in Figure 5.12. Both DGL and Bison parser are built with JavaCC parser generator version 4.0[4]. The parser and a non-trivial lexical analyzer for Bison grammars are both based GNU Bison version 2.3[5]. All matrix computations are performed with Java Matrix Package (JAMA) version 1.0.2[6]. Graph algorithms are implemented using JGraphT library version 0.7.3. Some other utility libraries are listed in Appendix A.

[4]See https://javacc.dev.java.net/ for more information about JavaCC.

[5]See http://www.gnu.org/software/bison/ for more information about GNU Bison.

[6]JAMA is a public-domain linear algebra package for Java developed by MathWorks and NIST. See http://math.nist.gov/javanumerics/jama/ for more information.

Package	*Description*
`thesis.poc.yaga.analysis`	Grammar analysis engine
`thesis.poc.yaga.grammar`	Grammar object model
`thesis.poc.yaga.parser`	Grammar parser infrastructure
`thesis.poc.yaga.parser.bison`	Implementation of Bison parser
`thesis.poc.yaga.parser.dgl`	Implementation of DGL parser
`thesis.poc.yaga.test`	Test classes

Figure 5.12: Structure of packages in YAGA implementation

Chapter 6

Related Work

6.1 Source Code Generation Methods

In the following sections we discuss several existing approaches to source code generation and relate them to the methods and techniques developed in this work sections. Note that most of our research presented in previous chapters was done independently of these results.

6.1.1 Dynamic Grammars

According to our research, the first work on the subject of source code generation was done by Hanford in 1970. In [17] he describes a so-called "syntax machine", a tool for generating random program for testing compiler front ends. It is an attempt to create a universal source code generator as discussed in Section 3.3. The tool takes as input a description of the target language expressed in a formalism of *dynamic grammars*. Dynamic grammars are defined as context-free grammars with addition of *syntax generators*. Syntax generators are basically actions performing self-modifications of the grammar. Hence the name "dynamic grammars". This mechanism should allow the syntax machine to deal with some of the context-sensitive rules (constraints) of the target language. The following quotations from the original paper [17] express the author's view.

> "The context-sensitive nature of programming languages derives from the fact that these languages allow declarations"
>
> "The basic idea in our formulation of these constraints it that the effect of writing a declarative statement in a program can be represented as a change to the context-free grammar."

While these statements were plausible at the time they were written, they are slightly outdated in the context of modern programming languages. Declarations are still an important feature of context-sensitive syntax, but there are other rules as well. As discussed in Subsection 4.5.4, dealing with the type system in a statically typed programming language seems to be the most difficult problem for a source code generator. It is not easy to address this problem in the context of dynamic grammars.

However, Hanford's syntax machine has other interesting features. It includes a restricted backtracking facility, possibility to set a limit on the number of recursive applications of a production rule and also weights on the production rules. The tool was implemented and applied in practice for ECMA Algol, FORTRAN IV and a major subset of PL/I.

6.1.2 Purdom's Algorithm

In [36] Purdom presented an algorithm that computes a minimal set of sentences which cover all production rules of a given context-free grammar. The algorithm is known in the literature as *Purdom's algorithm*. The main benefit of the Purdom's algorithm is that it produces a small finite set of sentences which have a good chance of adequately testing the parser of the target programming language. Since it is a deterministic algorithm, it always produces the same set of sentences for a given grammar. This poses a problem when the compiler under test "learns" its test case library.

However, the main limitation of the Purdom's algorithm is that it does not provide any means to handle context sensitive syntax rules of the target programming language. In [27, 28] Malloy and Power provide a modern reformulation of the Purdom's algorithm and discuss a practical application for programming languages C and C++. The authors acknowledge that "(...) if one or more of the automatically generated test cases is *syntactically or semantically* incorrect, then the confidence of structural "coverage" may be compromised" (emphasis added). The assumption of the algorithm is that testing with a set of sentences which utilize all productions rules of the underlying context-free grammar will "cover" all important logic of the parser. However, if some of the sentences are rejected because of violation of context-sensitive rules the assumption is no longer valid.

On the other hand, Purdom's the original paper was publish in 1972 and therefore served as basis for future work to several researches [6, 2, 33, 18].

6.1.3 EBNF with Actions

In 1980, Celentano et al. published a paper discussing a sentence generator for testing compilers [6]. The sentence generator is based on the Purdom's algorithm with minor modifications – see Subsection 6.1.2. The target programming language is described to the generator by "(...) an extended BNF grammar which can be augmented by actions to ensure contextual congruence, e.g. between definition and use of identifiers". By contextual congruence the authors refer to context-sensitive syntax rules. They correctly identify the most difficult problem related to context-sensitive source code generation.

> "The most difficult problems are the congruence between declarations and usage of identifiers, and the compatibility of identifier types within expressions"

The following quotation summarizes how context-sensitive syntax rules are handled in the presented sentence generator.

> "To ease the implementation, we assumed that actions cannot interfere with the sentence generation strategy; that is, actions may modify the already generated part of a test program (e.g. by inserting a declaration or an initialization), but they may not affect the CHOOSE and SELECT procedures of the generator."

This is exactly opposite to our approach described in Section 4.5. Our left-to-right AST generation principle assumes that the already generated part of the AST is fixed and tries to "interfere with the sentence generation strategy", i.e., influence rule selection procedures, to satisfy the context-sensitive rules. The approach of Celentano's sentence generator does not introduce impossible choice and pathological choice situations as described in Subsection 4.5.4.1, but it becomes problematic in context of a more complex programming language with lexical scoping, user-defined types and other advanced features. Celentano et al. implemented sentence generators for Olivetti Mini PL/I (a subset of PL/I), interpreter for Olivetti P6060 and a BASIC interpreter.

6.1.4 Attributed Test Grammars

Independently of the work of Celentano et al. [6] Duncan and Hutchison published an article in 1981 discussing a new method for automated generation of test cases. The method is based on the formalism of *attributed*

test grammars. These are a special modified version of attribute grammars developed by Knuth back in the 1960s [23, 24]. The grammar can generate test cases either randomly or systematically – see Chapter 3.

Attributed test grammars make use of synthesized, inherited and hybrid attributes. The attributes are used to guide the generation process. There are several mechanisms for manipulation with the attributes. They are called *guards*, *actions* and *designators*. Guards are used as context predicates on production rules of the grammar, actions are used for output and designators allow assigning values to attributes. In our view, these mechanisms essentially constitute a formally specified "programming language" which uses attributes instead of variables. However, it is rather simplistic language and using it to describe more complex context-sensitive rules seems to be either impossible or very inconvenient. Therefore we do not see any clear advantages in adopting attributed test grammars as opposed to directly embedding the specification of the target programming language into the implementation of the source code generator – as was done in case of PASGEN (see Section 5.2).

Note that the original paper [6] focuses only on the description of generation method and attributed test grammars. Implementation of an actual test case generator is discussed as an area for future work.

6.1.5 Context-free Parametric Grammars

Another way of describing a target programming language to a source code generator was proposed by Bazzichi and Spadafora in [2]. They present a source code generator that operates with a formalism of *context-free parametric grammars* which extend context-free grammars in a context-sensitive direction[1]. The generator is essentially based on Purdom's algorithm modified to work with context-free parametric grammars – see Subsection 6.1.2. The goal is to "(...) generate compilable programs for different programming languages, rapidly and cost-effectively" and also to "(...) generate incorrect programs in a controlled way".

The characterizing feature of context-free parametric grammars is that non-terminal symbols are allowed to have parameters. The range of each parameter is defined by a special language.

Example Let $V_T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ be the set of terminal symbols, and let $X = \{x, y, z, x_a, y_b, z_c\}$ be the set of languages $x = \{\mathbf{a}^n \mid n \geq 1\}$, $y = \{\mathbf{b}^n \mid n \geq 1\}$,

[1] The authors claim that one of the advantages of context-free parametric grammars is that they still retain the structure and readability of BNF notation.

$z = \{\mathbf{c}^n \mid n \geq 1\}$, $x_a = \{\mathbf{a}\}$, $y_b = \{\mathbf{b}\}$, $z_c = \{\mathbf{c}\}$. Let R be the following set of production rules

$$\begin{aligned} S &\rightarrow Ax_a y_b z_c \\ Axyz &\rightarrow Axx_a yy_b zz_c \\ Axyz &\rightarrow xyz \end{aligned}$$

The context-free parametric grammar $G = (X \cup \{S, A, \mathbf{a}, \mathbf{b}, \mathbf{c}\}, V_T, S, R)$ generates the context-sensitive language $L(G) = \{\mathbf{a}^n\mathbf{b}^n\mathbf{c}^n \mid n \geq 1\}$. A derivation of the string $\mathbf{a}^3\mathbf{b}^3\mathbf{c}^3$ looks like

$$\begin{aligned} S &\rightarrow Ax_a y_b z_c = A\mathbf{abc} \rightarrow \\ A\mathbf{abc} = Axyz &\rightarrow Axx_a yy_b zz_c = A\mathbf{aabbcc} \rightarrow \\ A\mathbf{a}^2\mathbf{b}^2\mathbf{c}^2 = Axyz &\rightarrow Axx_a yy_b zz_c = A\mathbf{a}^3\mathbf{b}^3\mathbf{c}^3 \end{aligned}$$

In context-free parametric grammars describing real programming languages, parameters are used to pass structures similar to a symbol table along the grammar. For example a sequence of tuples like

$$[ID, TYPEVALUE, TYPE, VTYPE, IDLIST, IDLISTV]$$

Parametric context-free grammars were used in practice to describe the programming language PLZ and a subset of Pascal. The generated programs were used for testing several compilers including Olivetti PLZ, PLZ/SYS by Milan Polytechnic and 4 different Pascal compilers: Stony Brook Pascal/360, IBM Pascal/VS, Pascal 8000 IBM 360/370 Version, Stanford Pascal Compiler.

6.1.6 Sentence Generator for PT

In [33], Murali and Shyamasundar present a language specific sentence generator for testing a compiler for PT, a subset of Pascal. They focus on generation of both correct and incorrect programs. The program correctness is considered with respect to the classification introduced by Celentano in [6] – see Figure 6.1.

According to the authors the most important features of the sentence generator are the following.

- The generation of lexically, syntactically and compile-time correct and incorrect programs.
- All the generated programs are indented.

1. totally invalid programs
2. lexically correct programs in which lexemes are well-formed
3. syntactically correct programs, which are correct with respect to the formal context-free grammar
4. compile-time correct programs, which are syntactically correct and satisfy all context-sensitive rules (dependencies)
5. run-time correct programs, which can be successfully executed
6. logically correct programs, which produce the expected results

Figure 6.1: Hierarchy of program correctness

- The number of statements in a generated program can be controlled.
- A limit can be placed on the number of binary operators in an expression and on the number of procedures in the generated program.
- The number of generations for each invocation of the system can be controlled.
- New test data can be provided each time in a test cycle (this is because the test generator uses a random selection scheme).
- Effective weights can be placed for the random selection scheme to emphasize testing particular constructs (thus avoiding the generation of long sequences to ensure the occurrence of the construct).

The sentence generator is implement using the recursive descent generation technique that we describe in Subsection 4.5.2.1. In this respect the generator is very similar to PASGEN. An example of a "generative procedure" is shown in the Figure 6.2. However, the sentence generator for PT generates the target source code directly and therefore diverges from the architecture discussed in Subsection 4.5.2. Our architecture suggests generating an abstract syntax tree first and then gradually transforming it to the textual representation.

The following quotation shows that Murali and Shyamasundar acknowledge existence of grammar neutral (language independent) generation techniques in line with our discussion of universal source code generators – see Section 3.3. We focus on language independent generation techniques based on context-free grammars in sections 4.1, 4.2 and 4.3.

```
procedure FACTOR
  begin
    case RANDOM(9) of
      0, 1: IDENTIFIER;
      2, 3: UNSIGNEDINT;
      4, 5:
        begin
          WRITE('''');
          repeat
            CHARACTER;
          until RANDOM(1) = 0;
          WRITE('''');
        end;
      6:
        begin
          IDENTIFIER;
          WRITE('['); EXPRESSION; WRITE(']');
        end;
      7:
        begin
          WRITE('('); EXPRESSION; WRITE(')');
        end;
      8: PREDECLARED;
      9:
        begin
          WRITE('NOT␣'); FACTOR;
        end;
    end; (* CASE *)
  end; (* FACTOR *)
```

Figure 6.2: Example procedure in the sentence generator for PT

> "(...) [Syntax] definition can be supplied to the system as input, in which case it is converted into a suitable representation and used in the generation (...) However, the [syntax] definition can also be embedded in the system itself, as in the case of the recursive descent parser. We use the latter method, and hence the sentence generator is tailored for PT"

The sentence generator for PT can operate in two modes: *context-free mode* and *context-sensitive mode*. An interesting feature of the context-free mode is the possibility to limit the number of binary operators in an expression. When a binary operator is generated the remaining limit is divided equally between the expressions constituting the operands. The unused operator limit carries forward from the first operand to the second and then to the higher level. In our context this technique would help to achieve the regularity criterion – see Section 3.2.

The context-sensitive mode of the sentence generator for PT allows an interesting comparison with PASGEN since both tools have to deal with similar problems. Both sentence generator and PASGEN adopt a similar approach when it comes to context-sensitive rules governing scopes and declarations. PASGEN uses the concept of inverse symbol tables described in Subsection 4.5.3 while the approach of the sentence generator is summarized in the following quotations.

> "When the declarative part of a program block is generated, the information of declared objects is maintained in a symbol table. The declarative part, therefore, creates a contextual environment within which the statements are generated."

> "If there are no variables within the scope, assignments statements are not generated. However, if there are variables, assignment statements may be generated"

The second quotations describes a very specific "hard-coded" solution to prevent the impossible choice situations in one context. We attempt to treat this problem in a more general way in PASGEN, particularly in relationship to the static typing rules. To deal with the typing rules, sentence generator for PT introduces another specific solution similar to the method of type constraints described in Subsection 4.5.4.1.

> "Since expression operands must be *type compatible*, operand information is supplied to the procedure that generates the expressions. While generating the operands, a variable of the operand

> type may have to be generated. In this case, the symbol table is examined to determine the existence of a variable of the required type. If such a variable is not available, an appropriate constant is substituted." (emphasis added)

We apply a similar approach in PASGEN for generating expressions. However, the assumption that a constant can always be generated instead of a variable depends on a programming language. It might be the case in PT, but it certainly does not work in many common high-level programming languages. For example it is not possible to write a constant of an array type or a record type in Pascal or in C. The following Pascal code excerpt demonstrates the problem.

```
type TMatrix = Array[1..100, 1..100] of Real;

function MatrixDeterminant(var A : TMatrix) : Real;
begin
  ...
end;


...
{ there is no variable of type TMatrix in the scope }
d := 1.5 * MatrixDeterminant( ??? );
```

Murali and Shyamasundar acknowledge that "[t]he system is capable of producing programs that humans do not think of when designing test cases" and therefore confirm our own observations – see Chapter 1. They further assert that the sentence generator was successfully used in the development of parsers and an intermediate code generator for PT.

Note that despite the apparent similarities, PASGEN – our prototype source code generator for Pascal – was developed independently of the results presented by Murali and Shyamasundar. See Section 5.2 for more detailed discussion of PASGEN.

6.2 SCFGs and Natural Language Processing

Stochastic context-free grammars (see Section 4.3.2 and Section 4.4) are studied extensively in the fields of computational linguistics and natural language processing. Together with probabilistic parsing algorithms stochastic grammars can be used for syntactic analysis of natural languages [3]. In this

line of research several methods and techniques for empirical estimation of stochastic grammars were developed[2]. It has been conjectured by Wetherell in [40] that these methods always yield *consistent* stochastic context-free grammars (see Subsection 4.4.1). A first result in this direction was presented by Chaudhuri et al. in [7]. He showed that if the probabilities of SCFG are calibrated to maximize the likelihood of a given set of sample parse trees then the grammar is always consistent. In later works by Sánchez and Benedí [37], Chi and Geman [11, 10] the result was extended with some limitations to expectation maximization methods which include the Inside-Outside algorithm and the Viterbi algorithm. Finally Nederhof and Satta proved in [34] that these estimation methods produce consistent grammars under most general conditions.

6.3 Other Related Work

Besides compilers, the source code produced by automatic generation methods can be used for testing other programing language processing tools. In [13] Daniel et al. describes an automated technique for testing refactoring engines. The core of their technique is a framework for iterative generation of structurally complex test inputs. They have applied their technique for testing Eclipse and Netbeans, two popular open-source IDEs for Java. As a result, several bug in both IDEs were exposed.

[2]Empirical estimation of SCFGs usually mean automated calibration of production rule probabilities starting from a given ambiguous context-free grammar

Chapter 7

Conclusion

The goal of this work was to develop methods and algorithms for generating non-trivial syntactically correct programs. The generated programs are intended to be used as input data for testing compilers or other source code processing tools. Traditional compiler test suites are composed of hand-written sample programs attempting to cover all interesting aspects of the target programming language. Creating such test suites is a difficult and time consuming task. As a result, high-quality test suites are either expensive or not available at all. We propose that source code generation methods can be used to automate construction of larger compiler test suites and thus make compiler testing easier and more affordable.

To put our research into a proper context, we have surveyed a substantial amount of literature related to the subject. Surprisingly, fruitful ideas have been found in many different fields of study including compiler construction, type theory, software testing and natural language processing. The most important topics and their relationship to our work are summarized in Chapter 6.

The first main result of this work is a set of theoretical methods and algorithms applicable for source code generation independently of the target programming language. When talking about programming languages, we differentiate between context-free and context-sensitive syntax. The set of methods developed in this work is divided accordingly.

We begin with a definition of an abstract sentence generation algorithm that models a context-free grammar derivation. The algorithm leaves out concrete descriptions of two important steps – *symbol selection* and *rule selection*. We propose two symbol selection methods (DFSS, BFSS) and three rule selection (RRS, PRS, FRS) methods. These can be plugged into the abstract sentence generation algorithm to produce different concrete algorithms. However, the randomized character of RRS and PRS methods introduces

the possibility of non-terminating computation that we call the *termination problem*. To deal with the termination problem PRS method transitions from plain context-free grammars to stochastic context-free grammars. It uses production rule probabilities to guide the rule selection process. In line with the seminal work of Booth and Thompson [5] we define a *consistent* stochastic grammar to be a grammar that reduces the probability of divergence to zero. It is desirable to use consistent grammars with the PRS method. Hence in Subsection 4.4.1 we present an algorithm for determining consistency of a given stochastic context-free grammar. The yes-or-no answer given by the algorithm is insufficient in some cases, particularly when we are manually calibrating probabilities in a larger grammar. Therefore we have also developed a more advanced version of the algorithm that also localizes the source of grammar inconsistency. The so-called *modular consistency* algorithm decomposes the grammar into special components and analyzes the consistency of each component separately. In addition, an alternative way to deal with the termination problem is provided by the FRS method. Together, these methods and algorithms provide a simple framework for building context-free sentence generators.

Next we turn our attention to context-sensitive syntax of programming languages. We propose an architecture for context-sensitive source code generators composed of the AST generation core and the backend phases. This allows us to think about the non-trivial problem of AST generation independently of other aspects. We have developed two basic design principles for building AST generators – *recursive descent generation* and *left-to-right generation* – and also some supporting concepts like inverse symbol tables. However, the most important contribution related to context-sensitive AST generation is the development of two methods for dealing with static typing rules – *type constraints* and *combinators*. They are presented in Subsection 4.5.4.

The second main result of this work is an implementation of three prototype tools that test applicability of the theoretical methods and algorithms mentioned before. Two of these tools are source code generators, the third one is a grammar analysis tool.

The first one, called GRAMGEN, focuses on the context-free syntax of programming languages. It is a universal pseudo-random sentence generator driven by stochastic context-free grammar of the target language. GRAMGEN implements the abstract sentence generation algorithm with breadth-first symbol selection method. It allows to dynamically switch between all three rule selection methods based on specific metrics, e.g. switch from PRS to FRS when the intermediate sentential form reaches given maximal length.

The second source code generator is called PASGEN. It is a language

specific generator for Pascal that shows how to deal with selected rules of context-sensitive syntax which also appear in other programming languages. These include rules governing declarations of constants, types and variables as well as references to these entities, lexical scoping rules and, most importantly, static typing rules. In addition a few other language-specific rules are handled in order to generate source code that is fully compliant with ISO/IEC 7185:1990 standard for Pascal [20].

The third prototype tool that was developed as a part of this work is called YAGA. It is a grammar analysis tool that implements algorithms for determining consistency of stochastic context-free grammars. The grammar can be given in one of the two supported formats – DGL and Bison. It is very convenient that YAGA can process Bison grammar definitions directly. Bison is still one of the most popular tools for building parsers and hence there is a large number of existing Bison grammars available for testing.

The broader scope of this thesis suggests that source code generation is rather an extensive topic. Even though we have explored a substantial portion of the subject matter, there are many paths open for future research. A potential line of future work could involve the integration of all source code generation methods that were developed in this work into a common system. The individual prototype implementations would also benefit from integration into a single tool since they are mostly complementary. However, there are several challenges related to the integration of context-free and context-sensitive source code generation methods. For example, it is not clear how to enforce context-sensitive rules in settings where derivation trees are not constructed according to the left-to-right principle (e.g. when breadth-first symbol selection is applied). Since we have not focused on optimizations, there certainly is some space for more efficient data structures and search algorithms in several areas like finalization rule discovery, modular consistency, inverse symbol tables, type constraints, combinators and others.

All the initial goals of this thesis were met, i.e., we have developed several methods for context-free and context-sensitive source code generation and implemented these methods in prototype tools. We believe that the results of this work can be used to build production-quality source code generators and therefore can indirectly contribute to the improvement of compiler reliability.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2nd edition, August 2006.

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

[3] Jose-Miguel Benedi and Joan-Andreu Sanchez. Estimation of stochastic context-free grammars and their use as language models. *Computer Speech & Language*, 19(3):249–274, 2005.

[4] David L Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[5] T.L. Booth and R.A. Thompson. Applying probability measures to abstract languages. *IEEE Transactions on Computers*, 22(5):442–450, 1973.

[6] A. Celentano, S. Crespi-Reghizzi, P.D. Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software - Practice and Experience*, 10(11):897–918, 1980.

[7] R. Chaudhuri, Son Pham, and O.N. Garcia. Solution of an open problem on probabilistic grammars. *IEEE Transactions on Computers*, 32(8):748–750, 1983.

[8] R. Chaudhuri and A. N. V. Rao. On a property of probabilistic context-free grammars. *International Journal of Mathematics and Mathematical Sciences*, 6(2):403–407, 1983.

[9] R. Chaudhuri and A. N. V. Rao. Approximating grammar probabilities: solution of a conjecture. *Journal of the ACM (JACM)*, 33(4):702–705, 1986.

[10] Zhiyi Chi. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1):131–160, 1999.

[11] Zhiyi Chi and Stuart Geman. Estimation of probabilistic context-free grammars. *Computational Linguistics*, 24(2):299–305, 1998.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.

[13] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley, Boston, MA, USA, 1995.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification (3rd Edition).* Addison-Wesley Professional, 2005.

[16] R. Hamlet. Random testing. In J.Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[17] K.V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[18] William Homer and Richard Schooler. Independent testing of compiler phases using a test case generator. *Software - Practice and Experience*, 19(1):53–62, 1989.

[19] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition).* Addison Wesley, 2nd edition, 2000.

[20] ISO/IEC Technical Committee JTC 1/SC 22. *ISO 7185:1990: Information technology – Programming languages – Pascal.* International Organization for Standardization, Geneva, Switzerland, 1990.

[21] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[22] Donald E. Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[23] Donald E. Knuth. Semantics of context-free grammars. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[24] Donald E. Knuth. The genesis of attribute grammars. In *Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[25] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In *The 18th IFIP International Conference on Testing Communicating Systems (TestCom 2006), New York City, USA, May 16-18, 2006*, volume 3964 of *LNCS*. Springer Verlag, 2006.

[26] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech & Language*, 4(1):35–56, January 1990.

[27] Brian A. Malloy and James F. Power. An interpretation of purdom's algorithm for automatic generation of test cases. In *Proceedings of 1st Annual International Conference on Computer and Information Science (ICIS '01)*, pages 310–317, Orlando, Florida, USA, 2001.

[28] Brian A. Malloy and James F. Power. A top-down presentation of purdom's sentence-generation algorithm. Technical Report NUIM-CS-TR-2005004, National University of Ireland at Maynooth, May 2005.

[29] Peter M. Maurer. Reference manual for a data generation language based on probabilistic context free grammars. Technical Report CSE-87-00006, University of South Florida, Tampa, FL 33620, 1987.

[30] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 07(4):50–55, 1990.

[31] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Software - Practice and Experience*, 22(3):223–244, 1992.

[32] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms (Cambridge International Series on Parallel Computation)*. Cambridge University Press, August 1995.

[33] V. Murali and R. K. Shyamasundar. A sentence generator for a compiler for pt, a pascal subset. *Software - Practice and Experience*, 13(9):857–869, 1983.

[34] Mark-Jan Nederhof and Giorgio Satta. Estimation of consistent probabilistic context-free grammars. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 343–350, New York City, USA, June 2006. Association for Computational Linguistics.

[35] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

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

[37] Joan-Andreu Sanchez and Jose-Miguel Benedi. Consistency of stochastic context-free grammars from probabilistic estimation based on growth transformations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):1052–1055, 1997.

[38] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. *SIGSOFT Software Engineering Notes*, 29(4):133–142, 2004.

[39] R. D. Tennent. *Principles of Programming Languages.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[40] C. S. Wetherell. Probabilistic languages: A review and some open questions. *ACM Computer Surveys*, 12(4):361–379, 1980.

[41] B.A. Wichmann. Some remarks about random testing. Technical note draft, National Physical Laboratory, May 1998.

Appendix A

Attached Software Components

The thesis includes a DVD disk containing the source code and executables of the three prototype tools described in Chapter 5. It also contain sample data and the text of the thesis itself. The disk has the following directory structure:

- `data/` – Sample inputs and outputs
  - `bison/` – collection of sample Bison grammars
  - `dgl/` – collection of sample DGL grammars
  - `generated-sources/` – examples of Pascal source code generated by PASGEN
- `programs/` – Binary packages of the prototype tools
  - `gramgen/` – GRAMGEN binaries for Microsoft Windows platform
  - `pasgen/` – platform independent PASGEN executables
  - `yaga/` – platform independent YAGA executables
- `sources/` – Source code of the prototype tools
  - `gramgen/` – source code of GRAMGEN
  - `pasgen/` – source code of PASGEN
  - `yaga/` – source code of YAGA
- `thesis/` – Text of this thesis in PDF format

A.1 Grammar-driven Generator (GRAMGEN)

GRAMGEN is a prototype sentence generator which generates sequences of symbols according to a given stochastic context-free grammars. The main executable can be found in the directory

```
<DVD_ROOT>/programs/gramgen/bin
```

It provides a very simple command line interface:

```
gramgen -g <file> -S <symbol> [-d <number>] [-f] [-h]
        [-l <number>] [-o <file>] [-s <number>]
```

The following table describes the individual command-line options.

Option	*Description*
`-d <number>`	Depth limit for the generated derivation tree
`-f`	Finish generation using finalization rules
`-g <file>`	Grammar definition file written in DGL format – see 5.1.2.2
`-h`	Print this help message
`-i`	Ignore production rule weights
`-l <number>`	Length limit on the generated sequence of symbols
`-o <file>`	Write generated sentence to a file
`-s <number>`	Seed number for the random number generator
`-S <symbol>`	Name of the non-terminal symbol from the specified grammar that will be used as the start symbol for sentence generation.

The following two examples show the output of GRAMGEN when executed on grammar `pascal01.dgl` with start symbol `program`, depth limit 10 and seed number 13. In the first example the generator is executed *without* finalization enabled.

```
$ gramgen -g ../../../data/dlg/pascal01.dgl -S program -d 10 -s 13

GRAMGEN - Grammar-driven Source Code Generator (version 1.0-alpha)
Charles University in Prague, Faculty of Mathematics and Physics

Initializing random number generator with seed [13]...
Parsing DGL grammar from [../../../data/dlg/pascal01.dgl]...
Running sentence generation engine...
  WARN: Depth limit reached (11/10).

Generated sentence (49 symbols):
------------------------------------------------------------------------
```

```
var a : Longint ; a : Real ; begin if ( ( ( expr + expr ) / b ) * c )
then c := a + ( expr / expr ) else i := ident - expr - expr * a ; end.
------------------------------------------------------------------------
Done.
```

Compare the generated sentence with the second example where the generator is executed *with* finalization enabled.

```
$ gramgen -g ../../../data/dlg/pascal01.dgl -S program -d 10 -s 13 -f

...
Running sentence generation engine...
  WARN: Depth limit reached (11/10).
  Switching to finalization rule selection mode...

Generated sentence (49 symbols):
------------------------------------------------------------------------
var a : Longint ; a : Real ; begin if ( ( ( a + a ) / b ) * c )
then c := a + ( a / a ) else i := a - a - a * a ; end.
------------------------------------------------------------------------
Done.
```

The design of GRAMGEN is described in Section 5.1. GRAMGEN was implemented in C++. It can be build with a included set of makefiles. The makefiles were tested with the MinGW port of GNU make version 3.81 and the MinGW GCC compiler 3.4.5. However, the makefiles are reasonably platform independent. They should work on other platform with minor modifications.

A.2 Pascal Source Code Generator (PASGEN)

PASGEN is a language-specific source code generator that attempts to produce syntactically correct Pascal programs according to the ISO/IEC 7185:1990 specification [20]. It is implemented in Java and therefore the binaries should be platform independent. To simplify the execution (e.g. setup of Java classpath) we provide launch scripts for the most common platforms, i.e., a batch file for Microsoft Windows and a shell script for UNIX. These are located in the directory

```
<DVD_ROOT>/programs/pasgen/bin
```

PASGEN provides the following command line interface:

```
pasgen [-c <file>] [-d <file>] [-h] [-o <file>] [-r <number>]
       [-s <number>] [-t <number>]
```

The individual command line options are described in the table below.

Option	*Description*
`-c,--config <file>`	Configuration file for the source code generator
`-d,--dump-tree <file>`	Dump abstract syntax tree of generated program to specified XML file
`-h,--help`	Print this help message
`-o,--output <file>`	Write generated program to file (default: "generated.pas")
`-r,--retry <number>`	Maximal number of retries when generation fails
`-s,--seed <number>`	Seed number for random number generator
`-t,--timeout <number>`	Number of seconds until generator is forced to stop (default: 10 seconds)

Note. The predefined probabilities of the production rules embedded in the structure of the generator are not calibrated to from a consistent stochastic grammar. Although some heuristic methods are applied to prevent this, in some cases the generator follows an infinite derivation sequence. Therefore an explicit time limit is always set for the generation process. It can be controlled by the `--timeout` option. When the allocated time limit is exceeded, it is assumed that the generator has entered an infinite derivation sequence and hence the generation process is terminated.

The following example shows the expected console output of PASGEN when it is executed with seed number 13.

```
$ pasgen --seed 13

PASGEN - Pascal Source Code Generator (version 1.0-alpha)
Charles University in Prague, Faculty of Mathematics and Physics

Starting generator with seed [13]...

Writing generated program to file [generated.pas]...
Done.
```

External Libraries

PASGEN uses several open-source utility libraries to simplify some of the implementation tasks.

Commons CLI is a library that helps with the definition and parsing of command-line options. Our implementations use the version 1.1 of the library. It the most recent version from the 1.x branch to date. For more information about Commons CLI see http://commons.apache.org/cli.

Commons Lang is a collection of utility classes that supplements some of the standard JDK functionality. Commons Lang is also used by Commons CLI. The version 2.3 of this library is used by all implementations. See also http://commons.apache.org/lang/.

Log4J is a logging library used by both PASGEN and YAGA. The most recent version of the library to date from the stable 1.2 is used, i.e. the version 1.2.15,. Log4J can be downloaded from http://logging.apache.org/log4j.

XStream is an object-to-XML serialization library. PASGEN uses version 1.2.2 of this library to dump IR tree to a specified XML file on request. Note that none of the core functionality of PASGEN depends on this library. For more information about XStream see http://xstream.codehaus.org/.

Note. PASGEN should not be executed directly from the DVD disk. because it will fail when it attempts to write debugging messages to a log file and the generated program to an output file. Both the log file and the output file are placed by default into the current working directory. However, the current working directory is write-protected when it is located on a DVD disk.

A.3 Grammar Analyzer (YAGA)

YAGA is a grammar analysis tool that implements algorithms for determining consistency of stochastic context-free grammars. The grammar can be given in one of the two supported formats – DGL and Bison. The grammar format and the grammar definition file name are the only required options in the command line interface provided by YAGA. We also provide launch scripts for YAGA similar to those provided for PASGEN. They are located in the directory

```
<DVD_ROOT>/programs/yaga/bin
```

When the launch scripts are used YAGA has the following command-line syntax:

```
yaga [-d <file>] -f <format> -g <file> [-h] [-i] [-o <file>]
```

The individual command line options are described in the table below.

Option	*Description*
`-d,--dump <file>`	Dump annotated grammar to file or stdout after analysis (default: stdout)
`-f,--format <format>`	Format of grammar definition ("dgl" or "bison")
`-g,--grammar <file>`	Grammar definition file
`-h,--help`	Print this help message
`-i,--ignore`	Ignore production rule weights from the grammar definition (use default value "1")
`-o,--output <file>`	Write analysis report to file

The example below shows the expected console output of YAGA when it is asked to analyze the grammar `pascal01.dgl`. For comparison we recommend running YAGA on the same grammar with the `--ignore` option. Optionally the analysis reports for both executions can be observed by adding appropriate `--output` options.

```
$ yaga -g ../../../data/dlg/pascal01.dgl -f dgl

YAGA - Yet Another Grammar Analyzer (version 1.0-alpha)
Charles University in Prague, Faculty of Mathematics and Physics

Parsing DGL grammar from [pascal01.dgl]...
Normalizing production weights to probabilites...
Running grammar analysis...
```

```
  Start symbol          : [program]
  Terminal symbols      : [32]
  Non-terminal symbols  : [18]
  Grammar consistency   : [true]
  Derivability components : [12]
  Inconsitent components : [0]

Done.
```

External Libraries

YAGA also uses several external open-source libraries. The matching versions are used for the libraries that also appear in PASGEN. This is the case of Commons CLI 1.1, Commons Lang 2.3 and Log4J 1.2.15. The following additional libraries are used by YAGA:

JAMA is a basic linear algebra package for Java. It provides user-level classes for constructing and manipulating real matrices. YAGA use the version 1.0.2 of the library. JAMA is developed by MathWorks and NIST. For more information about JAMA see http://math.nist.gov/javanumerics/jama/.

JGraphT is a free Java graph library that provides mathematical graph-theory objects and algorithms. YAGA relies on the version 0.7.3 of this library. JGraphT can be downloaded from http://jgrapht.sourceforge.net/.

Appendix B

Generated Source Code Samples

This appendix contains several examples of the Pascal source code generated by PASGEN. All presented programs can be found in the directory `data/generated-sources` on the included DVD disk.

Listing B.1: generated-01.pas

```
program GeneratedProgram;
const
    CONST1 = - 0.5035517;
    CONST2 = 4.0;
    CONST3 = 6.0;
    CONST4 = - CONST3;
    CONST5 = - 28;

type
    T1 = ^T2;
    T2 = boolean;

var
    o, i1, z, c : real;
    j2, index, bb, k, var1 : integer;
    m, v : ^T2;
    var2 : ^integer;

begin
    k := var1;
    index := bb div index div 22 + ((var2^)) div (CONST5 - var2^) * var2^
        * 4 - var2^ div (22 div (k div (((var2^) div 2))));
end.
```

Listing B.2: generated-02.pas

```
program GeneratedProgram;

const
    CONST1 = + 28;
    CONST2 = 8.0;

type
    T1 = boolean;
    T2 = ^integer;
    T3 = char;
    T4 = integer;
    T5 = real;
    T6 = T5;

var
    var1 : integer;
    var2, z, var3, var4, j4 : ^integer;
    bb, d, b, e : ^integer;
    o, var5 : integer;
    var6, t, var7, u : T5;

begin
    e^ := ((- 11 div z^ div (var1) + 4));
    var7 := 26 div o + (((o + (1795987108 div (CONST1 div (var1 div CONST1
        ) * (var5 + (+ e^)) + (var3^ - var4^ * CONST1)))) - (CONST1) div
        (((var5) - (z^ - CONST1) + o div (o + var4^ div CONST1) div CONST1
        ) div o)) * var5 + ((+ (CONST1 + var1 - (var5 * var1 div b^ - var4
        ^ - b^ div var4^ - var1 + (+ 8 - (var1 div (CONST1 div ((var1 + ((
        o + var2^) div CONST1) + var5 div (CONST1))) div var1) + (var3^))
        div (var5 - (16 div (- var1) div var5 div 5 div d^ div (- var5 div
         28 div var5)) div var2^)) - (((d^) div (var2^ div CONST1 * var1))
         div CONST1 - var5)) - e^ - CONST1) * (17) div var5 div CONST1) *
        0.0) - 23 div CONST1);

end.
```

Listing B.3: generated-12.pas

```
program GeneratedProgram;

type
    T1 = ^T2;
    T2 = char;

var
    var1, s, z, var2, c : record
        var3 : integer;
        var4 : boolean;
        j2 : record
            g, tmp, m : ^T2;
            n, i, f : integer;
            i1, d : ^T1;
            i3, var5 : T1;
        end;
        u, var6, l : record
            a, b : ^char;
            y : boolean;
            var7, var8, var9 : ^T1;
            t, q : T2;
        end;
    end;
    p, j1, var10, var11, j4 : T2;
    e, v, h : record
    end;

begin
    case 'q' of
        'Z' :
            z.u := c.u;
        'I', '9' :
            v := e;
        '6' :
            z.j2.g := nil;
        '$' :
            with h, s, c.l do
                s.j2.m := nil;
        'T', ']' :
            var2.j2.m := var2.j2.g;
        'j', '0' :
            z.j2.d^ := var2.j2.var5;
    end;

    with s.var6, h do
        if s.var6.y then
            repeat
                v := v;
```

```
            while var1.var4 do
                z.u.var7 := (var1.var6.var7);
            z.j2.i1^ := var2.j2.var5;
        until c.var4 and not var2.var4;
with c.u do
    z.var4 := var2.j2.i + s.j2.f <> 23 div 27 - var2.j2.n;
z.j2.i := (var1.var3 * z.j2.f) - 3;

if var2.u.y or not var2.var4 or (0.72384113 - var1.var3 = 21) or (var1
    .var4) and z.var4 then
    s.j2.i1^ := z.u.var7^
else
    begin
        z.j2.i1 := nil;
        with var1 do
            var2.var6.var7 := nil;
        repeat
            s.j2 := (c.j2);
            var2.var6.var7 := nil;
            var2.var6.y := 3.0 <> 447550891;
            s.l.var8 := var2.var6.var9;
            with h, z.u, c.j2 do
                c.l.y := c.var4;
        until (var2.u.y or s.var4 or z.u.y);
        if s.var6.y and var2.u.y then
            var1.j2.var5 := s.j2.d^;
        var1.j2.i1 := nil;
    end;

e := v;
h := e;
z.u.b := (c.u.a);
c.j2.i1 := s.j2.i1;

begin
    repeat
        if var2.var4 and s.var4 then
            begin
                s.j2.n := - c.j2.i * 2;
                var2.j2.g := var2.j2.tmp;
                repeat
                    c.j2.tmp := z.j2.m;
                    var1.var6.y := 'z' >= (p);
                    s.l.var7 := nil;
                    var2.j2 := var1.j2;
                    c.u.y := c.var6.y and var2.var6.y or (c.var4);
                until + ((c.j2.f)) <= var1.j2.f;
                with s.l do
                    var2.j2.var5 := c.j2.i3;
```

```
            end
        else
            c.u.t := z.l.a^;

        repeat
            s.l.q := 'L';
            var1.j2.tmp := nil;
            s.var6.a := (nil);
            var1 := c;
            repeat
                with c.j2 do
                    if var2.var6.y and var1.var4 or z.var6.y or z.var4
                        or var2.var4 and var1.var6.y then
                        z.j2 := c.j2
                    else
                        var2.j2.i3 := c.j2.d^;
                c.j2.i1 := nil;
                var2.l.b := nil;
                while s.j2.n mod z.var3 <> c.j2.i * z.j2.i - z.j2.i *
                    0.3298955 do
                    begin
                        z := c;
                        var2.var3 := + 28;
                    end;
                var1.l := s.var6;
            until c.var4 or c.var4 >= (c.l.y and c.l.y) or var2.var4;
        until z.var4;

        z := (var2);
        while not z.var4 do
            var2.j2.d^ := (s.j2.i3);

    until s.l.y and c.var6.y or var2.var4 and s.var4;

    var2.var6.var8^ := var2.j2.d^;
    var2.u := var2.l;
    var2 := s;
    z.j2 := c.j2;
    s.l.var7 := s.var6.var7;
end;
end.
```