groo

programming
language

Abstract

This report documents the groo programming language. This includes design decisions, syntax analysis, contextual analysis and execution of a groo program.

A formal operational semantics is given for the groo language as well as a formal description of the type system.

A lexer and parser generator has been implemented to conduct the syntax analysis, and a type checker to conduct the contextual analysis.

Finaly a recursive interpreter and a virtual machine, which can be used to execute groo programs, have been implemented.

Theme:
Tools, Languages and Compilers

Project period:
DAT2, spring semester 2010

Project group:
D201A

Concluded:
28th May 2010

Group members:
Karsten Jakobsen
Anne K. Jensen
Jonas F. Jensen
Sabrine C. H. Mouritsen
Thomas S. Nielsen
Lars Kærlund Østergaard

Supervisor:
Hans Hüttel

Prints: 7

Report page count: 129

Appendix page count: 8

The department of computer science at Aalborg University

Preface

The reader is expected to be familiar with operational semantics and set theory in order to understand the formal language specification. The implementation of groo is programmed in C++, so familiarity with this is necessary to understand the code.

The notation given in Hüttel [2010] has been used to describe the operational semantics and type system of groo.

Obtaining the Source Code and Documentation

The CD provided with this report contains source code, documentation, as well as code samples and binaries. Instructions are provided in the `README` file located at the root directory of the CD. The CD includes a makefile which generates sources and builds the groo compiler. GNU Coreutils, GCC, Python are required to run this makefile.

Contents

List of Figures

Part I

Introduction

CHAPTER 1

Project Description

1.1 Problem Analysis

Today, a lot of software is written in slow interpreted dynamic scripting languages - especially web applications. These, rather slow, dynamic languages are chosen because they are easy to use, offer platform independence and high productivity. However, when these applications need to scale, it is often done by merely connecting more hardware to the system, so the application can accommodate more traffic. If, instead, the code were written in more efficient languages or statically compiled, the performance could be improved significantly. It would probably still be necessary to increase the amount of hardware in use, in order to scale sufficiently - but the amount of extra hardware required could be reduced considerably. In addition, these languages do not lend themselves well for large-scale web applications - the following text will explore why.

The obvious question is: if statically typed languages are so much faster, why are they not in wider use - in place of dynamic languages? A possible answer is that dynamic scripting languages may offer more productivity gains up-front and therefore offer quicker development cycles. They may also have a less steep learning curve than more explicit statically typed languages. Despite the fact the scripting languages are dynamically typed, they still offer some degree of safety, since the most common languages (e.g. Python, PHP, Ruby) provide no means of directly manipulating pointers. In terms of productivity we refer strictly to the amount of time required in order to create a working program. A script may take half as long to write as a C/C++/Java program, as shown in the phonecode programming problem in Prechelt [2002].

Nevertheless, productivity is still a loosely defined method of measurement, as it does not tell us anything about the quality of the code written, as there are cases where a dynamically typed program may contain many errors that are not immediately apparent, due to the typing mechanism and possible lack of code coverage during execution. Neither does it tell us how many development cycles are required for a program to become stable. Short development cycles are advantageous for rapid prototyping. Yet, when a dynamically typed program makes the transition from prototype to production code, a lot of discipline is required of the programmer.

It is fair to assume that a programmer, being human, may overlook common errors or less obvious errors in a large code base. The error-checking burden is actually placed on the programmer, as they must be able to foresee many execution errors. Reasoning about program behaviour obviously requires a lot of run time testing and the enforcement of disciplined use of existing modules relies very much on the programmer. Clearly, the productivity gained from the writeability of scripting languages is minimised when these factors are taken into account. Therefore it is in its place to conclude that productivity is affected by testability and maintainability.

In terms of performance, scripting languages fall short of their statically typed counterparts. For example, the typical memory consumption of a script is about twice that of a C/C++ program. Also, scripting languages are less reliable than statically compiled languages - many errors are first discovered at run time, rather than at compile time. Types in static languages are explicitly declared and checked at compile time, which means that all type errors will be found at compile time.

In dynamically checked languages the type of a variable is not declared during writing; the actual values assigned to a variable decides its type. Also, type checking is performed at run time. Thus, variables can refer to a value of any type during execution. For example, assigning an arbitrary non-integer value to a variable, the programmer intended to use as an integer later in the program, is perfectly legal. However, this error will only be uncovered in the running program. As a project created in a scripting language grows in complexity, bugs will be introduced at some point - yet the programmer has only very limited error-recovery tools at hand. Development with a scripting language may also be more error prone, and more time will often be spent debugging in dynamically typed languages compared to static languages.

A very popular scripting language for web applications is PHP, which offers high productivity, yet suffers performance-wise for larger applications. PHP is a dynamic, weakly-typed, and interpreted language, which consequently will never perform as fast as compiled code. Recently Facebook, a popular social networking site built with PHP, started transforming its PHP source code into highly optimized C++, which is then compiled with g++ into native code by the means of a tool called HipHop for PHP. This tool includes a code transformer as well as a reimplementation of the PHP run time system. By using this technology Facebook reduced their average CPU consumption by fifty percent [Zhao, 2010].

Considerations for a Programming Language

To begin with, a well-designed programming language makes use of the fundamental concept of abstraction. With regard to language design, abstraction can be thought of as initially identifying all syntactic categories of the language and subsequently designing a coherent set of abstraction facilities for each of these. For instance, many languages support process abstraction with sub-programs, expression abstraction with functions, abstract data types, objects and modules.

Furthermore, the principle of data type completeness is a feature found in well-designed languages. This basically means that all data types are first class without arbitrary restriction on their use. A language is type complete if and only if all three of the following conditions are met:

- Each identifier, operator and expression has a type and its type may not depend on the context in which it appears.
- For each type in the language it is possible to write an expression having this type.
- Function parameters must be able to be of any type and return results of any type. So, it is possible to yield a function of an even more complex type.

An example of a language that breaks this principle is an early version of Pascal, where it was not possible to yield an array type from a procedure [Tennent, 1981, Demers and Donahue, 1980].

The aforementioned design guidelines still apply to the domain of web programming, since web applications can be treated as complex software systems which benefit from being implemented in well-designed programming languages. Moreover, the programming domain of web software comprises many different languages for each their purpose. One important goal is to deliver dynamic web content, which requires various computational tasks to be performed on the server, rather than the client (browser) - such as serving a specific document or requesting data from a database and outputting it to a mark-up language (specifically HTML/XHTML). Many web programming languages are designed to be directly embedded in an HTML document. Typically these are scripting languages, for example PHP or Python. Besides simply outputting HTML to the client, these languages can execute other programs on the server or request services from external sources, which can be written in another language.

There are many factors to consider when designing a programming language for web applications. Among these factors is readability - which is rather broadly defined as how easily a program can be read and comprehended by a human. Readability also has a direct relation to the maintainability of applications written in a specific language. Since readability is such a broad criterion, it makes sense to consider it in the context of the domain of web applications, in order to get a more clear understanding of which things are important for readability in this particular area. Therefore, the syntax must be designed to fit the domain, in order to avoid obscure and long-winded code.

Another aspect that affects readability is the simplicity of the language. Specifically, this could be measured by the amount of basic constructs available in the language. It is important to strike a balance between simplicity and complexity, though, since either extreme can result in overly complicated or verbose code [Sebesta, 2008].

To sum up, some goals for a programming language can be readability, reliability, and efficiency. These features can be supported by implementing a type system, for instance. Other very important aspects, such as maintainability and code reuse, can be facilitated by using object-oriented programming, especially in conjunction with a type system. In the following we will discuss how a language can benefit from a type system and object-orientation.

The Advantages of a Type System

A type system is a formal method used to prove the absence of certain program errors and ensure general correctness properties in a programming language. A type system can be implemented in many different ways, though its purpose is usually the same; a contract between the programmer and the compiler. A type system captures the programmer's intentions and, most importantly, captures execution errors in advance. Pierce [2002] gives the following definition:

> *A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.*

This means that a type system can also be regarded as a tool for reasoning about programs. This method places emphasis on the classification of terms with respect to the properties of the values they will compute when a program is executed.

A type is basically an invariant which can be used to restrict the set of values a variable can hold during execution. Such an invariant can, for example, be specified by explicitly enhancing a variable with a type annotation.

To verify that a program is type correct, a type checker is used. A static type checker rejects potentially unsafe programs at compile time. This means that good program behaviour is determined before execution. A program that is accepted by the type checker is said to be well-typed; conversely a rejected program is ill-typed.

Uncovering errors at compile time rather than run time has tremendous implications for program safety. The types of errors can be divided into two categories: trapped errors that cause computation to halt immediately and the more subtle untrapped errors that may cause arbitrary program behaviour. The latter class may go unnoticed, and cause potentially disastrous problems without crashing the program. A language is said to be safe if no program fragments cause untrapped errors to occur.

Some possible execution errors may be characterized as forbidden errors, which are a set of predetermined execution errors. They should include all untrapped errors as well as a subset of the trapped errors. A language is weakly checked if its set of forbidden errors does not contain all untrapped errors. In other words; not all unsafe operations are detected statically. For example, C/C++ has a lot of unsafe features, such as pointer arithmetic, which places a lot of responsibility on the programmer to enforce safety.

The safety of a type system must be judged according to its own set of forbidden errors, which is the definition of which kinds of behaviour it aims to prevent. In other words, the set of possible errors are decided based on the language's definition of run time type errors. However, there is considerable overlap between the behaviours regarded as run time type errors in many different languages. So, there exists common ground for judging and classifying what is bad program behaviour.

A type system can offer more than simply prevention of low-level errors. It can be used to protect the integrity of data abstractions by enforcing information hiding. For example, illegal access to protected fields will be treated as a run time type error.

Static type checkers calculate a static approximation of type correctness, since they evaluate type information at compile time. They can only prove the absence of certain errors, and not their presence, which makes static type checking conservative. A type system is yet incomplete, since some well-behaved programs will be rejected by a static type checker.

For example, a program fragment may contain an if-then-else-statement of the form, *if C then E1 else E2*, where *C* is a boolean condition, *E1* is a well-typed expression and *E2* is an ill-typed expression. The condition *C* could be expressed as $1 == 1$ so that it always evaluates to *true* at run time. Even so, this program fragment cannot be statically determined to be well-typed, since static analysis cannot predict that the *E1* branch will always be executed. Nevertheless, this kind of behaviour is beneficial, since it ensures that all programs are well-typed, regardless of how frequently branches may be executed at run time.

Types have many advantages, which are discussed in the following.

They can enhance readability, since explicitly typed languages can provide variables, functions, etc., with type annotations, which give a program some degree of documentation.

A type system makes the job of a programmer easier as it can be built into a compiler, enabling automatic type checking.

Both reliability and efficiency can be improved with types, since types provide a safety guarantee, as all type errors are uncovered during compilation with a static type checker. This prevents unsafe programs from ever running. Also, many routine programming errors can be captured, making debugging easier. Typed languages also permit certain optimisations, such as cheaper memory allocation for variables, since the execution overhead of run time checks can be avoided.

As mentioned earlier, a typed language may enforce program modularity and the integrity of data abstractions. This may allow a typed program to be organized into interfaces for program modules which can be compiled independently of each other. The programmer can decide what information a given module's interface should expose. This enables loose coupling between modules, since the dependency between modules is limited to their respective interfaces. This kind of information hiding greatly supports code reuse as dependencies between code fragments are minimized. In addition, when the interfaces are stable, changes to a certain module do not affect its consuming modules, thus avoiding recompilation of these [Cardelli, 2004, Palsberg and Schwartzbach, 1994, Pierce, 2002].

Benefits of Object-Oriented Programming

As mentioned above, object-oriented programming (OOP) supports maintainability and code reuse. In addition, concepts, such as information hiding and modularity, can be implemented straightforwardly using this model. OOP introduces the idea of objects, classes, late binding, and sub-classing.

A class can be regarded as a template from which objects can be constructed. An object is an encapsulated state and classes describe objects with the same implementation. Objects created from a certain class are instances of this particular class. This allows a program to contain a number of instances of a certain class, enabling the use of one class on different data at the same time.

A class can contain procedures which are called methods. A method enables an object to receive messages or send messages to other objects. For example, a message could be passed to an object in the following manner: `obj.m(x)`, where `obj` is the object with method `m`, or message selector, which takes the argument `x`, which too is an object. The concept of message passing between objects is a central part of OOP.

In addition, OOP employs late binding. Late binding basically means that when a message is sent to an object its implementation is dynamically bound, depending on the type of the receiving object. For example, we may require a procedure to render a list of different HTML elements, say headings and paragraphs, which inherit from a common class, called `Element` containing a method called `Print`. The two classes override this method, so instances of `Heading` print `"<h1>...</h1>"` and `Paragraph` instances print `"<p>...</p>"`. As we iterate over a mixed collection of `Elements` we send the `Print` message to each instance. `Element.Print` will then dynamically invoke the correct implementation of `Print` belonging to the current object [Palsberg and Schwartzbach, 1994].

Sub-classing

With regard to code reuse it is very useful to be able to generalise certain data structures or objects, which share a common number of operations, yet need to behave in a different way or be extended. In addition, restricting the usage of these objects is important for safety and for ensuring that a program behaves as desired. It is possible to implement these requirements with the means of sub-classing.

Sub-classing is a technique for reusing object templates, namely classes. A sub-classing mechanism known as inheritance can be employed. With inheritance new classes that share the implementation or specification of an existing class can be created, effectively allowing reuse of an implementation. The inheriting class is called a subclass, whereas the class from which it inherits is its superclass. It is possible to apply sub-classing to any number of levels. This means that a code base can be organised into logically grouped class hierarchies, which produce the benefits of reusable software, including more organized code.

Depending on the particular predicate and sub-classing mechanism used, various degrees of restriction on the reuse of objects can be enforced. Here, types are needed in order to force disciplined reuse of objects, since they serve the purpose as predicates on objects. There are a number of different predicates which can be used. They differ in which and how many aspects of objects are under consideration and whether they put emphasis on the implementation or the specification.

When classes are used as types the predicate requires that an object is an instance of a certain class or any of its subclasses. However, the way this is determined relies on the specific sub-classing mechanism. In the following we will describe a number of different sub-classing mechanisms, proposed by Palsberg and Schwartzbach [1992]. The list is ordered by how restrictive the requirements are.

- **Class types + arbitrary subclasses**: Methods may be added, overwritten or removed. This is the least restrictive predicate, since any collection of methods correspond to some subclass.
- **Name compatibility**: Demands that there exists a particular set of named methods.
- **Interface types**: We consider name compatibility and include the types of the arguments of the required methods.
- **Class types + monotone subclasses**: New methods can be added and existing method bodies can be overridden. This still preserves the interface.
- **Method behaviour**: Restrictions are imposed on the behaviour of the required methods. This could be done by specifying pre- and post conditions.
- **Class types + strictly monotone subclasses**: It is only possible to add new methods. In addition, the specified behaviour must have a certain implementation.

The methods listed above offer different ways of restricting the possibilities for object reuse and limit type compatibility by specifying a predicate that an object must satisfy and thereby a relation between implementations. Moreover, they propose four different definitions of how types are defined, namely by class + subclasses, name compatibility, interfaces, or behaviour [Palsberg and Schwartzbach, 1992].

Sub-typing

In the previous section we reviewed different models for the semantics of sub-classing, including which transformations on classes each model allowed. In the following the concept of sub-typing is discussed.

Sub-typing signifies a relation on types. Generally this relation is regarded as a partial order and is defined by the type system. For example, if T_1 is a subtype of T_2, written as $T_1 <: T_2$, then any object of type T_1 is also an object of type T_2. Therefore any object of type T_1 can be substituted as an argument where an object of type T_2 is expected.

Sub-typing essentially enables us to use the subtype polymorphism of objects. As mentioned in the previous section, exactly how sub-typing should be understood depends on how a type is defined. The definition of a type must be sound, i.e. it ensures the absence of certain run-time errors, as discussed earlier. We essentially require that sub-typing provides adequate protection by the subtypes of the types of the formal parameters. Again, Palsberg and Schwartzbach [1992] introduces four major notions of sub-typing. The different definitions of a type and their corresponding sub-typing mechanisms are listed below.

- **Class types + sub-classes**: subclassing
- **Name compatability**: more methods
- **Interface**: conformance
- **Behaviour**: weaker preconditions, stronger postconditions

When classes are used as types, we end up with a type system that is sound by definition. Class types + subclasses are regarded as implementation types, whereas the rest are specification types.

With specification types, sub-classing and sub-typing can be two very distinct concepts. For name compatibility, a sub-type must simply implement more methods, otherwise it is trivially the same type as its super-type. An interface sub-type must conform to its super-type. Specifically, the sub-type must respect name compatibility, which is further constrained by requiring that the required named methods have conforming signatures.

Finally, for behaviour types, a sub-type must have weaker pre- and stronger postconditions.

In essence, one must decide whether to separate classes and types, or rather, use specification types or implementation types or both at the same time. A type system based on classes requires that all instances of a given class have the same type. Specification types do not require that this rule is satisfied, because class relationships do not need to have anything in common with type relationships.

Each approach has its drawbacks. For instance, with implementation types we cannot convey if a class has more than one type and we do not permit two different type stacks of an implementation to be interchangeable. Specification types exhibit another set of potential problems. In the case that interface types are used, conceptually dissimilar types may correspond to each other.

1.2 Problem Statement

Since we have discussed some of the main differences between dynamic languages and static languages, we would like to find out if it is possible to encourage the creation of more efficiently written web applications. Moreover, it would be interesting to see if it is feasible to produce a statically typed object-oriented language with the features that developers have come to expect from the dynamically typed languages, while maintaining the performance benefits of compiled languages. In the following we have listed some of the questions we would like to answer.

- How can we create a statically typed object-oriented programming language which can be used for web development?
- Can we implement abstraction facilities similar to those that exist in popular dynamically typed languages, while preserving the advantages of a statically typed programming language?
- How can such a language be formalised?
- How can we implement a type system that enables sub-classing?
- How can we translate and execute code written in this language?

To begin with will need to take various language design aspects into consideration, including syntax, abstraction, and type completeness. Furthermore, we will have to design a type system in order to create a safe, readable, and efficient programming language. We can formalise both the type system and semantics of this language by the means of structural semantics. In addition, we have highlighted some of the benefits of object-oriented programming, which will be included in the language. This obviously requires us to decide on what kind of sub-classing technique we need. This goes hand-in-hand with the choice of sub-typing mechanism.

Initially, this will require us to translate the program text into a more machine-friendly format called an abstract syntax tree. This can be accomplished with custom made or existing lexers and parsers. The abstract syntax tree representation will undergo a number of transformations, such as type checking, so that programs can be executed according to the specifications.

For our language we will use a type system based on implementation types, specifically class + subclasses, where the sub-typing mechanism is based on subclasses. The reason for this choice is primarily due to the benefits of code reuse provided by class types. Finally, we will employ some degree of implicit typing to find a balance between readability and writeability of programs. With this we will explore the idea of finding a good compromise between explicit and implicit type information. Some degree of implicit typing may also be advantageous for developing rapid web prototypes.

In order to test our language we will develop an interpreter. We will create a simple virtual machine, in order to compile programs into a faster intermediary format, which will be easier to reason about in terms of performance.

Overview of this Report

The following shortly describes the contents and structure of this report.

Project Description

Part I contains the problem analysis, problem statement and an informal language specification.

Semantics

Part II contains the abstract syntax, the operational semantics and the type system.

Syntax Analysis

Part III explains the lexer and parser generator implementation, and how the generated lexer and parser transform text into an AST.

Contextual Analysis

Part IV covers AST decoration and the implementation of a type checker.

Execution

Part V documents how groo is executed using a recursive interpreter, and how groo programs are translated into the intermediate language, gril, including the execution of this on the virtual machine VROOM.

Closing

Part VI discusses how groo could be improved in future work, our results and concludes the report.

Appendices

Part VII

CHAPTER 2

Language Design

This chapter discusses the language design decisions for the programming language.

2.1 Design Requirements

The primary goal is to create a language which lends itself well to web development purposes. The language should be able to scale from small-scale prototypes to larger and more complex web applications. More specifically, it should be relatively easy to start with a small web page and transition to a more elaborate web application.

The goal for our programming language, which we have named groo, is to mimic dynamic languages such as Python and Ruby, with faster execution time and more correct code. The requirements that we deem important for groo are listed below:

Productivity The ability to write well-typed code efficiently with a writeable syntax.

Code reuse The ability to easily reuse code, by generalising facilities into classes.

Encapsulation The ability to create data abstractions and protect an implementation by enforcing disciplined use of its interface.

Type safety A statically checked language provides better run-time safety and more guarantees for program correctness.

Efficiency Static type checking allows efficient memory allocation and faster execution, since many run-time checks are unnecessary.

2.2 Type System

As mentioned in the problem statement we have decided to base the type system on implementation types. This is because it enables code reuse, and requires explicit type definitions. The type system for groo is *nominal*, since the name of a class conveys its type and sub-typing is explicitly declared.

Explicit subtype declaration avoids spurious categorisation, where a structurally compatible type can be substituted in a place where a logically different type is expected. This prevents the case where, a class, C, cannot suddenly be replaced by the logically unrelated type B, just because it coincidentally happens to conform to the specification of B [Pierce, 2002].

Apart from classes groo will have primitive types, which are `int`, `bool` and `float`. In this way we may limit the message passing overhead for primitive expressions. If, for instance, `int` were an object and one were to multiply two integers with this syntax: `a * b`, one would typically send `b` to the message selector `*` on the object `a`. Rather than doing this, primitives will be treated separately from objects. Of course, this decision results in a language that is not purely object-oriented, however this inconsistency may be more efficient.

Moreover, we have decided to support anonymous functions, first-class functions, returning functions (higher-order functions) or members as values, which requires us to support closures. In terms of type equivalence for function types, specification types will be used. So, for two given functions f and g, the two are equivalent if and only if both their argument types and return types are the same. Syntactically, function types are not assigned an identifier, but are denoted by their return and argument types.

Finally, we will support a tuple abstraction in groo. The primary reason for including tuples is to allow functions to yield multiple return values and avoid output parameters which may be complicated to use. In fact, the use of output parameters is discouraged in the .NET framework, because the general audience cannot be expected to master output parameters [Abrams and Cwalina, 2008, section 5.8.3].

2.3 Informal Language Specification

groo will borrow some inspiration from the syntax of the popular web development language, Python. This choice is influenced by the overall writeability, readability and succinctness of Python code. Of course, we cannot escape from the fact that the syntax of a language may very much be influenced by subjective arguments. Nevertheless, the language has certain useful abstraction facilities, such as tuples, iterators and easy string manipulation, which are exposed in a minimalistic way through certain syntactical productions. These may aid us in designing the required abstraction facilities for groo with a desirable syntax.

Blocks

We have decided to use significant whitespace to indicate blocks, i.e. a block is a continuous set of statements with the same indentation. We believe that significant whitespace discourages large complex code blocks, rather it encourages the developer to split complex routines into smaller subroutines.

Type Annotations

To improve writability we have decided to use implicitly typed variables. The declaration of a variable is indicated by the keyword `var`, e.g. declaring the variable x could have the form `var x = 0`. Later assignment to the variable x is written as `x = 1`. This means that variables will be implicitly typed. The type of the right-hand-side expression yields the static type of the declared variable.

Members of classes (instance variables and methods) require explicit type annotation. This design choice serves both for the purpose of documentation and to provide guidance for the type checker. Type annotated members increase readability, since the reader does not have to study the block of a method to know its return type.

The motivation for having implicitly typed variables and explicitly typed members is an attempt to balance readability and writability. One could easily argue against implicitly typed variables in some cases. For instance, the yielded type of the statement `var x = <complex expr>` could be unclear.

However, since the type of the expression is determined compositionally, meaning that the type of an expression depends only on the types of its immediate constituents, one can deduce the actual type by studying its immediate constituents. This minor problem may be remedied by using parenthesises to group complex sub-expressions. An integrated development environment may also provide extra assistance by annotating variables with type information gathered from automatic type checking.

Scope

groo will have static scope rules. With static scope the name of a variable refers to its local lexical environment and variable occurrences are matched to their binding statically. The parameter mechanisms available in the language will be call-by-reference for objects and call-by-value for primitive types.

Code Sample

A code sample for the groo language is shown in listing 2.1.

This program uses the class `HtmlBuilder` to construct different kinds of HTML tags. This is done with the `maketag` and `maketagAttr` methods, which each return anonymous functions that return a string representing an HTML element with its contents. The method signatures in line 22 and 27 show that each method yields a function type. This is indicated by the prefixed type annotation resembling $T_1 \to T_2$.

So, the variables `h1` and `p` are actually functions which output content enclosed by their respective HTML elements. In line 12 An anonymous function which outputs links/anchors is created, called and passed to the `p` function. The result is a link which is nested within a paragraph.

Listing 2.2 shows the output.

More code samples can be found in the appendix in part VII.

```
class MainClass:
    int main():
        out(”<html>\n<head></head>\n<body>\n”)
        var html = HtmlBuilder()

        var h1 = html.maketag(”h1”)
        var p = html.maketag(”p”)

        out(h1(”Hello World!”))
        out(p(”A paragraph.”))
        out(p(”Another paragraph.”))
        out(p(html.maketagAttr(”a”)(”Click here”,”href=\’foo.com\’”)))

        out(”</body>\n</html>\n”)
        return 0

    void out(string txt ):
        prints (txt )

class HtmlBuilder:

    string−>string maketag(string name):
        var tag = (string txt )−>string:
            return (”<”+name+”>”+txt+”<”+name+”/>\n”)
        return tag

    (( string,string)−>string) maketagAttr(string name):
        var tag = (string txt , string attr )−>string:
            return (”<”+name+” ”+attr+”>”+txt+”<”+name+”/>\n”)
        return tag
```

Listing 2.1: Code example for a simple webpage.

Listing 2.2: Output from running the code in listing 2.1.

```
<html>
<head></head>
<body>
<h1>Hello World!<h1/>
<p>A paragraph.<p/>
<p>Another paragraph.<p/>
<p><a href=′foo.com′>Click here<a/>
<p/>
</body>
</html>
```

CHAPTER 3

Infrastructure

The groo compiler is comprised roughly of four parts. When code is supplied for interpretation, each part handles a specific aspect of converting the code into a running program.

Lexical Analysis

The first part is the lexical analysis. The lexer reads the code provided to the interpreter, and splits the code into lexical chunks, or tokens, using the methods described in chapter 7. The lexer follows an instruction set looking for phrases that are known to it. The lexer is run on demand by the parser, which is the next step, or simply by groo if tokenize is run.

Syntax Analysis

The second step is the syntax analysis. This is where the parser constructs an abstract syntax tree (AST). It does this by attempting to place the tokens it requests from the lexer in the grammar it supports. When the parser completes a grammar rule, the parser creates a new node in the AST, thus effectively converting the user written code into a tree of different nodes describing concisely their purpose. How this is done is explained in chapter 8. However, while the code may have made perfect sense from a grammatical point of view, it may not make any actual sense. For instance, the parser does not distinguish between different types. It merely deduces that they are a Type, and moves on. It would therefore be perfectly legal to assign an integer to a variable declared as a string from the parsers point of view. Once the parser reaches an accept state, the contextual analysis begins.

Contextual Analysis

The contextual analysis is comprised of two visitors. One visitor handles error terminals that may have been created by the parser, if it suddenly receives a token it did not expect. These error terminals are caused by the user writing something syntactically incorrect. The other visitor is more interesting and handles type checking. The type checker visitor decorates the AST with type information. While it does this, it also checks to ensure that the AST satisfies the type rules, printing errors when type errors occur. The type checker and the contextual analysis in general is explained in detail in part IV. This decorated AST is then used later, during the code generation. Finding errors in either the error finder visitor, or the type checker visitor prevents the interpretation of the code.

Interpretation

After the program has passed the contextual analysis, the code can be interpreted. An interpreter executes a source program immediately without first translating it to another language. Both recursively and iteratively interpretation is supported, where recursively interpretation interprets the AST and iterative interpretation interprets a bytecode made from the AST.

The recursive interpreter is explained in chapter 10 and the iterative interpreter is explained in chapter 11.

Part II

Semantics

CHAPTER 4

Abstract Syntax for groo

In this chapter we will present an abstract syntax used to describe both the operational semantics and the type system (static semantics). An abstract syntax is a simpler version of the concrete syntax, where the main focus is show the structure of of the language.

Notice that this abstract syntax will be used to describe both the operational semantics and the static semantics. Though, some syntactic categories will not be relevant for both, e.g. $type$ annotations are only relevant in the static semantics.

Table 4.1 lists the syntactic categories, and table 4.2 and 4.3 list the abstract syntax for groo.

4.1 Syntactic categories in groo

n	$\in$ **Num**	*Numerals*
d	$\in$ **Var**	*Variables*
e	$\in$ **Expr**	*Expressions*
S	$\in$ **Stmt**	*Statements*
id	$\in$ **ID**	*Identifiers; names of classes, members, and types*
$decls$	$\in$ **Decls**	*Class or enum declarations*
$groo$	$\in$ **Program**	*Class and enum declarations of a groo program*
$members$	$\in$ **Members**	*Field and method declarations*
$labels$	$\in$ **Labels**	*enum declarations*
ve	$\in$ **VarExpr**	*Sequence of variables. Subset of Expr*
$param$	$\in$ **Param**	*Formal parameters*
arg	$\in$ **Arg**	*Arguments - actual parameters*
op	$\in$ **Op**	*Operators*
$type$	$\in$ **Type**	*Type annotations*

Table 4.1: Syntactical categories for groo.

4.2 Abstract syntax

In table 4.2 and 4.3 the productions for the abstract syntax can be found. In this abstract syntax we use semicolon to denote newlines, which are used to seperate statements in the concrete syntax. In order to facilitate variable declarations within statements in the type system, no sequantial statement production is available. Instead all statements, except return- and empty-statement, is followed by another statement, which is the next statement.

It should also be noted that in the concrete syntax, indentation, denoted with tabs, is used mark blocks.

$groo$	::=	$decls$	*groo*
$decls$	::=	$\textbf{class}\ id\textbf{:}\ members;\ decls$	*ClassDecl*
	$\mid$	$\textbf{enum}\ id\textbf{:}\ label;\ decls$	*EnumDecl*
	$\mid$	ϵ	*EmptyDecl*
$members$	::=	$type\ id;\ members$	*FieldMember*
	$\mid$	$type\ id\ \textbf{=}\ e;\ members$	*FieldMemberExt*
	$\mid$	$type\ id\ \textbf{(}\ param\ \textbf{):}\ S;\ members$	*MethodMember*
	$\mid$	ϵ	*EmptyMember*
S	::=	$\textbf{var}\ d\ \textbf{;}\ S$	*DeclStmt*
	$\mid$	$e\ \textbf{;}\ S$	*ExprStmt*
	$\mid$	$\textbf{if}\ e\textbf{:}\ S_1\ \textbf{;}\ S_2$	*IfStmt*
	$\mid$	$\textbf{if}\ e\textbf{:}\ S_1\ \textbf{else:}\ S_2\ \textbf{;}\ S_3$	*IfElseStmt*
	$\mid$	$\textbf{while}\ e\textbf{:}\ S_1\ \textbf{;}\ S_2$	*WhileStmt*
	$\mid$	$\textbf{return}\ e$	*ReturnStmt*
	$\mid$	ϵ	*EmptyStmt*
d	::=	$id\ \textbf{=}\ e\ \textbf{,}\ d$	*VarDecl-1*
	$\mid$	$id\ \textbf{=}\ e$	*VarDecl-2*
	$\mid$	$id\ \textbf{,}\ d$	*VarDecl-3*
	$\mid$	id	*VarDecl-4*
e	::=	$e_1\ \mathrm{op}\ e_2$	*BinaryExpr*
	$\mid$	$\mathrm{op}\ e$	*UnaryExpr*
	$\mid$	ve	*VarExpr*
	$\mid$	$ve\ \textbf{=}\ e$	*VarAssignExpr*
	$\mid$	$e_1\textbf{(}arg\textbf{)}$	*CallExpr*
	$\mid$	$e_1\textbf{( )}$	*CallExpr*
	$\mid$	$\textbf{(}\ param\ \textbf{)->}\ type\ \textbf{:}\ S$	*AnonymExpr*
	$\mid$	$\textbf{(}\ e\ \textbf{)}$	*ParenExpr*
	$\mid$	n	*NumExpr*
	$\mid$	**True**	*TrueLiteral*
	$\mid$	**False**	*FalseLiteral*
ve	::=	id	*VarAccess*
	$\mid$	$e\ \textbf{.}\ id$	*VarAttAccess*

Table 4.2: Abstract syntax for groo.

$labels$	::=	$id; labels$	*UnassignedLabel*
	$\mid$	$id = n; labels$	*Label*
	$\mid$	ϵ	*EmptyLabel*
$param$	::=	$type\ id, param$	*SeqParam*
	$\mid$	$type\ id$	*Param*
	$\mid$	ϵ	*EmptyParam*
arg	::=	e, arg	*SeqArg*
	$\mid$	e	*Arg*
	$\mid$	ϵ	*EmptyArg*
$type$	::=	$type$ **->** $type$	*FunctionType*
	$\mid$	id **,** $type$	*TupleType*
	$\mid$	id	*SimpleType*
op	::=	$+\mid - \mid * \mid/\mid << \mid >> \mid\%$	
	$\mid$	$\|\mid$ **or** $\mid\&\&\mid$ **and** $\mid!\mid$ **not** $\mid == \mid <= \mid >= \mid < \mid > \mid$!=	

Table 4.3: Abstract syntax for groo, continued.

CHAPTER 5

Operational Semantics

In this chapter we will formalise language design decisions for executing a groo program. The semantics will be described by giving a transition system for each syntactic category in the syntax. For this we will need to define some environments and auxiliary functions.

5.1 Environments

When evaluating a production we will need to save the result so it can be reused later. For this we introduce the environment-store model, as described below.

In the environment-store model, we have an environment, env, that is a partial map from identifiers to locations, and a store, sto, which is a partial map from locations to values. The set of all environments, $\mathbf{Env}$, and the set of all stores, $\mathbf{Sto}$, are defined below.

$$\begin{aligned} l \in \mathbf{Loc} &= \mathbb{Z} \\ env \in \mathbf{Env} &= \mathbf{ID} \rightharpoonup \mathbf{Loc} \\ sto \in \mathbf{Sto} &= \mathbf{Loc} \cup \{next\} \rightharpoonup \mathbf{Values} \cup \mathbf{Loc} \end{aligned}$$

In the definition of $\mathbf{Sto}$ $next$ is a special element that maps to the next unused location. Elements of $\mathbf{Env}$ are denoted by the metavariable env, likewise l is a metavariable for elements of $\mathbf{Loc}$ and sto is a metavariable for elements of $\mathbf{Sto}$. Values mapped to by $\mathbf{Sto}$ are defined as follows:

$$\begin{aligned} \mathbf{Primitive} &= \mathbb{Q} \cup \{True, False\} \\ \mathbf{FuntionValue} &= (\mathbf{Stmt} \times \mathbf{Param} \times \mathbf{Env}) \cup (\mathbf{Members}, \mathbf{Env}) \\ \mathbf{Objects} &= \mathbf{Env} \\ v \in \mathbf{Values} &= \mathbf{Primitive} \cup \mathbf{FunctionValue} \cup \mathbf{Objects} \end{aligned}$$

A primitive can be a rational number or a boolean value $True$ or $False$. A function value can be a tuple of a body, parameters and an environment or a tuple of members and an environment. In both cases the environment is the bindings known at the time of the declaration of the function. A tuple of members and an environment is a constructor. An object is a partial mapping of identifiers of its members to locations, thus an environment. An element of the set of all values, $\mathbf{Values}$, is denoted by the metavariable v.

By storing code in values, it is possible to do forward declarations, which makes describing mutual recursion simple. For example when declaring methods it is possible to declare the method before storing it, thus an environment where all other methods have been declared can be stored in the tuple of members and an environment, facilitating mutual recursion.

5.2 Auxiliary Functions

5.2.1 The numeral function

The numeral function is used to get the value of a numeral, and is defined as: $\mathcal{N} : \mathbf{Num} \to \mathbb{Q}$. For any numeral it returns the equivalent integer or rational number.

5.2.2 Update Function for Environments

Updating an environment env, so that the identifier id maps to the location l, is denoted $env[id \mapsto l]$ which is defined as:

$$env(id') = \begin{cases} env(id') & \text{if } id \neq id' \\ l & \text{if } id = id' \end{cases}$$

5.2.3 Update Function for Stores

Updating a store sto, so that the location l maps to the value v, is denoted $sto[l \mapsto v]$ which is defined as:

$$sto(l') = \begin{cases} sto(l') & \text{if } l \neq l' \\ v & \text{if } l = l' \end{cases}$$

5.2.4 The New Location Function

We introduce the auxiliary function $new : \mathbf{Loc} \to \mathbf{Loc}$ to find the next unused location given an unused location l. Since $\mathbf{Loc} = \mathbb{Z}$, we define $new(l) = l + 1$. The new function is used to update the $next$ element of sto, whenever we need to use a new location.

5.2.5 The Apply Operator

When evaluating a binary or unary operation described in the abstract syntax, we will introduce an apply operator. Passing an operator and one or two values result in the value after applying the operator. The definition of *Apply* can be seen in table 5.1.

5.3 Transition Systems in groo

Transition systems are used to define an operational semantics. A transition system is defined by a set of configurations and a set of transitions, also called the transition relation.

Table 5.1: Apply operator for expressions

$$\begin{align}
Apply(+, v_1, v_2) &= v_1 + v_2 \tag{5.1}\\
Apply(-, v_1, v_2) &= v_1 - f v_2 \tag{5.2}\\
Apply(*, v_1, v_2) &= v_1 \cdot v_2 \tag{5.3}\\
Apply(/, v_1, v_2) &= \frac{v_1}{v_2} \tag{5.4}\\
Apply(<<, v_1, v_2) &= v_1 \cdot v_2 \cdot 2 \quad \text{if } \{v_1, v_2\} \subseteq \mathbb{Z} \tag{5.5}\\
Apply(>>, v_1, v_2) &= \lfloor \frac{v_1}{v_2 \cdot 2} \rfloor \quad \text{if } \{v_1, v_2\} \subseteq \mathbb{Z} \tag{5.6}\\
Apply(\%, v_1, v_2) &= v_1 \bmod v_2 \tag{5.7}\\
Apply(||, v_1, v_2) &= v_1 \vee v_2 \tag{5.8}\\
Apply(\mathbf{or}, v_1, v_2) &= v_1 \vee v_2 \tag{5.9}\\
Apply(\&\&, v_1, v_2) &= v_1 \wedge v_2 \tag{5.10}\\
Apply(\mathbf{and}, v_1, v_2) &= v_1 \wedge v_2 \tag{5.11}\\
Apply(==, v_1, v_2) &= v_1 = v_2 \tag{5.12}\\
Apply(! =, v_1, v_2) &= v_1 \neq v_2 \tag{5.13}\\
Apply(<, v_1, v_2) &= v_1 < v_2 \tag{5.14}\\
Apply(< =, v_1, v_2) &= v_1 \leq v_2 \tag{5.15}\\
Apply(>, v_1, v_2) &= v_1 > v_2 \tag{5.16}\\
Apply(> =, v_1, v_2) &= v_1 \geq v_2 \tag{5.17}\\
Apply(-, v) &= -v \tag{5.18}\\
Apply(!, e) &= \neg v \tag{5.19}\\
& \tag{5.20}
\end{align}$$

A transition system is a triple $(\Gamma, \rightarrow, T)$ of configurations, Γ, end configurations, T, and a set of transitions, $\rightarrow$. T is a subset of Γ.

In the following sections we will define the transition systems for the relevant categories from section 4.1.

groo Program, Groo

A groo program is just a number of class and enum declarations succeeded with a call to the method $main$ in the class $MainClass$. The declarations are evaluated in an empty environment and store, and the classes and enums are then declared in these. Then the call $MainClass().main()$ is evaluated in the environment and store resulting from the declaration transistions, which will run the program.

The transition system for a groo program, $(\Gamma_{\mathbf{Groo}}, \rightarrow_{groo}, T_{\mathbf{Groo}})$, is therefore defined by

$$\Gamma_{\mathbf{Groo}} = \mathbf{Decl} \cup \mathbf{Env} \times \mathbf{Sto}$$

$$T_{\mathbf{Groo}} = \mathbf{Env} \times \mathbf{Sto}$$

and transitions are of the form

$$\langle decl \rangle \rightarrow_{groo} \langle env', sto' \rangle$$

and $\rightarrow_{groo}$ is defined as in table 5.2.

Class and Enum Declarations, Decl

When a class is declared the result is an identifier in the environment that points to a location where a constructor for the class can be found. A constructor is a tuple of $members$ and an env wherein other classes and enumerations have been declared. By storing the constructors in sto and declaring them in env, they can be declared before they are stored. Thus, mutually recursive constructors are easily achieved. Note that the type system ensures that constructors are not overwritten.

When an enum is declared the result is an updated environment, where the identifier for the enum points to a location where an object for the enum can be found. This object is an env from label identifiers to label contants, as defined in section 5.3. By declaring enums this way there is no need to introduce any special rules for handling enums in expressions.

The transition system for declarations, $(\Gamma_{\mathbf{Decl}}, \rightarrow_{decl}, T_{\mathbf{Decl}})$, is defined by

$$\Gamma_{\mathbf{Decl}} = \mathbf{Decl} \times \mathbf{Env} \times \mathbf{Sto} \cup \mathbf{Env} \times \mathbf{Sto}$$

$$T_{\mathbf{Decl}} = \mathbf{Env} \times \mathbf{Sto}$$

Transitions are of the form

$$\langle decl, env, sto \rangle \rightarrow_{decl} \langle env', sto' \rangle$$

and $\rightarrow_{decl}$ is defined as in table 5.3.

Class Members, Members

Members can either be fields or methods. Member transitions are used to create an instance of a class. Declaration results in an updated environment where the identifier of the members points to a location where the value of the member can be found. Both fields and members are stored as values on the object. The type system will prevent methods from being overwritten, so there is no need to take this into account here.

When a field is being declared the value will be the value yielded from evaluating an expression, if an expression has been provided. If not, there will not be a value at the location the identifier points to. If a method is being declared the value will be a function value, which is a tuple of the body of the method, the parameters, and the environment that every member of the class is declared in.

All members are declared before the values are stored, which enables mutual recursion.

The transition system for members,$(\Gamma_{\mathbf{Members}}, \rightarrow_m, T_{\mathbf{Members}})$, is defined by

$$\Gamma_{\mathbf{Members}} = \mathbf{Member} \times \mathbf{Env} \times \mathbf{Sto} \cup \mathbf{Env} \times \mathbf{Sto}$$

$$T_{\mathbf{DeclC}} = \mathbf{Env} \times \mathbf{Sto}$$

Transitions are of the form

$$env'' \vdash \langle members, env, sto \rangle \rightarrow_m \langle env', sto' \rangle$$

and $\rightarrow_m$ is defined as in table 5.4.

Statements, Stmt

A statement can both declare new variables, change the value of a variable - through an expression - and return a value. The return statement can stop the execution of the next statement.

$(\Gamma_{\mathbf{Stmt}}, \rightarrow_s, T_{\mathbf{Stmt}})$ is defined by

$$\Gamma_{\mathbf{Stmt}} = \mathbf{Stmt} \times \mathbf{Env} \times \mathbf{Sto} \cup (\mathbf{Values} \cup \epsilon) \times \mathbf{Env} \times \mathbf{Sto}$$

$$T_{\mathbf{Stmt}} = (\mathbf{Values} \cup \epsilon) \times \mathbf{Env} \times \mathbf{Sto}$$

Transitions are of the form

$$\langle S, env, sto \rangle \rightarrow_s \langle v, env', sto' \rangle$$

and $\rightarrow_s$ is defined as in tables 5.5 and 5.6.

Variable Declarations, Var

Variable declarations create new variables in the environment. There is no type annotation, as this will be inferred at the first comparison. A variable declaration can either have or not have an expression assigned to it. If it has, the expression is evaluated and the value is stored at a the newly created location for that variable. If not, only the location is created. Hence the type system for variable declarations, $(\Gamma_{\mathbf{Var}}, \rightarrow_d, T_{\mathbf{Var}})$ is defined by

$$\Gamma_{\mathbf{Var}} = \mathbf{Var} \times \mathbf{Env} \times \mathbf{Sto} \cup \mathbf{Env} \times \mathbf{Sto}$$

$$T_{\mathbf{VarExpr}} = \mathbf{Env} \times \mathbf{Sto}$$

Transitions are of the form

$$\langle d, env, sto\rangle \rightarrow_d \langle env', sto'\rangle$$

and $\rightarrow_d$ is defined as in table 5.8.

Expressions, Expr

The result of evaluating an expression is a value and perhaps a changed store. Expressions need to know the environment to look up variables and methods, but cannot declare new variables.

The instantiation of a class is classified as an expression. Evaluating the name of the class as an expression will yield the function value of that class. This contains the members of the class and the environment the class was declared in. Instantiating a class will therefore result in declaring all members of the class

A method call is also classified as an expression. Evaluating the expression results in a function value containing the body of the method, the parameters and the environment of the class. The arguments of the method call and the parameters given when the method was declared are evaluated together (See section 5.3) resulting in an environment and store containing the parameters with the values of the arguments. The body of the method call then evaluated in this environment and store results in a value, and a changed env and sto. But the environment is discarded so that variables declared in the method cannot be used outside the method call.

The transition system for expressions, $(\Gamma_{\mathbf{Expr}}, \rightarrow_e, T_{\mathbf{Expr}})$, is defined by

$$\Gamma_{\mathbf{Expr}} = (\mathbf{Expr} \times \mathbf{Sto}) \cup (\mathbf{Values} \times \mathbf{Sto})$$

$$T_{\mathbf{Expr}} = \mathbf{Values} \times \mathbf{Sto}$$

Transitions are of the form

$$env \vdash \langle e, sto\rangle \rightarrow_e \langle v, sto'\rangle$$

and $\rightarrow_e$ is defined as in table 5.9 and 5.10.

Variable Expressions, VarExpr

A variable expression can either be a sequence of variables and method calls followed by an id or just an id. The sequence is an expression, and hence gives an object value. Evaluating a variable expression returns a location for that id in the environment from the object value. $(\Gamma_{\mathbf{VarExpr}}, \rightarrow_{ve}, T_{\mathbf{VarExpr}})$ is defined by

$$\Gamma_{\mathbf{VarExpr}} = (\mathbf{VarExpr} \times \mathbf{Sto}) \cup (\mathbf{Loc} \times \mathbf{Sto})$$

$$T_{\mathbf{VarExpr}} = \mathbf{Loc} \times \mathbf{Sto}$$

Transitions are of the form

$$env \vdash \langle ve, sto \rangle \rightarrow_{ve} \langle l, sto' \rangle$$

and $\rightarrow_{ve}$ is defined as in table 5.11.

Enum Labels, Label

Enum labels are constants of an enum. An enum constant can either be assigned a value or not. However, as every constant needs a value one will be assigned automatically. This is done by a giving every transition a value. When declaring an enum the value, 0, is passed along with the transition, and this value is then incremented when an new constant is declared. As constants must be in sequence, a variable assigned by the programmer needs to be greater than the number of unassigned constants between it and the last-named constant. Constants cannot be assigned the same value. The transition system for enum labes, $(\Gamma_{\mathbf{Label}}, \rightarrow_{l}, T_{\mathbf{Label}})$, is then defined by

$$\Gamma_{\mathbf{Label}} = \mathbf{Label} \times \mathbf{Env} \times \mathbf{Sto} \times \mathbf{Values} \cup \mathbf{Env} \times \mathbf{Sto}$$

$$T_{\mathbf{Label}} = \mathbf{Env} \times \mathbf{Sto}$$

Transitions are of the form

$$v \vdash \langle label, env, sto \rangle \rightarrow_{l} \langle env', sto' \rangle$$

and $\rightarrow_{l}$ is defined as in table 5.12.

Arguments and Parameters, ArgParam

Arguments and parameters are evaluated in one transition, so that the values of the arguments can be stored at the correct locations of the parameters. An evaluation of an argument and a parameter subsequently change both the environment and the store. $(\Gamma_{\mathbf{ArgParam}}, \rightarrow_{ap}, T_{\mathbf{ArgParam}})$ is defined by

$$\Gamma_{\mathbf{ArgParam}} = \mathbf{Arg} \times \mathbf{Param} \times \mathbf{Env} \times \mathbf{Sto} \cup \mathbf{Env} \times \mathbf{Sto}$$

Table 5.2: Big-step Semantics for groo

$$[\text{groo}]\quad \frac{\langle decl, \{\}, \{\}\rangle \rightarrow_{decl} \langle env', sto''\rangle \\ env'' \vdash \langle members, env'', sto''\rangle \rightarrow_{m} \langle env^{(3)}, sto^{(3)}\rangle \quad env^{(4)} \vdash \langle S, sto^{(3)}\rangle \rightarrow_{s} \langle v, sto'\rangle}{\langle decl\rangle \rightarrow_{groo} v}$$

$$env'(\text{mainClass}) = (members, env'')$$
$$env^{(3)}(\text{main}) = (S, \epsilon, env^{(4)})$$

$$T_{\mathbf{ArgParam}} = \mathbf{Env} \times \mathbf{Sto}$$

Transitions are of the form

$$env' \vdash \langle arg, param, env, sto\rangle \rightarrow_{ap} \langle env', sto'\rangle$$

and $\rightarrow_{ap}$ is defined as in table 5.13.

Table 5.3: Big-step Semantics for Declarations

[ClassDecl]

$$\frac{\langle decl, env'', sto'' \rangle \rightarrow_{decl} \langle env', sto^{(3)}, v \rangle}{\langle \textbf{class}\ id\colon members;\ decl, env, sto \rangle \rightarrow_{decl} \langle env', sto' \rangle}$$

where $l = sto(next)$
$sto'' = sto[next \mapsto new(l)]$
$env'' = env[id \mapsto l]$
$sto' = sto^{(3)}[l \mapsto (members, env')]$

[EnumDecl]

$$\frac{\begin{array}{c} 0 \vdash \langle label, env'', sto'' \rangle \rightarrow_{l} \langle env^{(3)}, sto^{(3)} \rangle \\ \langle decl, env'', sto^{(4)} \rangle \rightarrow_{decl} \langle env', sto' \rangle \end{array}}{\langle \textbf{enum}\ id\colon label;\ decl, env, sto \rangle \rightarrow_{decl} \langle env', sto' \rangle}$$

where $l = sto(next)$
$sto'' = sto[next \mapsto new(l)]$
$env'' = env[id \mapsto l]$
$sto^{(4)} = sto^{(3)}[l \mapsto env^{(3)}]$

[EmptyDecl] $\langle \epsilon, env, sto \rangle \rightarrow_{decl} \langle env, sto \rangle$

Table 5.4: Big-step Semantics for Members

[FieldMember]

$$\frac{env^{(3)} \vdash \langle members, env'', sto'' \rangle \rightarrow_m \langle env', sto' \rangle}{env^{(3)} \vdash \langle id_t \ id; members, env, sto \rangle \rightarrow_m \langle env', sto' \rangle}$$

Where $l = sto(next)$
$env'' = env[id \mapsto l]$
$sto'' = sto[next \mapsto new(l)]$

[FieldMemberExt]

$$\frac{\begin{array}{c} env^{(3)} \vdash \langle members, env'', sto'' \rangle \rightarrow_m \langle env', sto^{(3)} \rangle \\ env^{(3)} \vdash \langle e, sto^{(3)} \rangle \rightarrow_e \langle v, sto^{(4)} \rangle \end{array}}{env^{(3)} \vdash \langle id_t \ id{=}e; members, env, sto \rangle \rightarrow_m \langle env', sto' \rangle}$$

Where $l = sto(next)$
$env'' = env[id \mapsto l]$
$sto'' = sto[next \mapsto new(l)]$
$sto' = sto^{(4)}[l \mapsto v]$

[MethodMember]

$$\frac{env^{(3)} \vdash \langle members, env'', sto' \rangle \rightarrow_m \langle env', sto'' \rangle}{env^{(3)} \vdash \langle id_t \ id(param) :S; members, env, sto \rangle \rightarrow_m \langle env', sto' \rangle}$$

Where $l = sto(next)$
$sto' = sto[next \mapsto new(l)]$
$env'' = env[id \mapsto l]$
$sto' = sto''[l \mapsto \langle S, param, env' \rangle]$

[EmptyMember] $env' \vdash \langle \epsilon, env, sto \rangle \rightarrow_m \langle env, sto \rangle$

Table 5.5: Big-step Semantics for Statements

[DeclStmt]

$$\frac{\langle d, \mathit{env}, \mathit{sto}\rangle \to_d \langle \mathit{env}'', \mathit{sto}''\rangle \quad \langle S, \mathit{env}'', \mathit{sto}''\rangle \to_s \langle \mathit{env}', \mathit{sto}'\rangle}{\langle \text{var } d; S, \mathit{env}, \mathit{sto}\rangle \to_s \langle \epsilon, \mathit{env}', \mathit{sto}'\rangle}$$

[ExprStmt]

$$\frac{\mathit{env} \vdash \langle e, \mathit{sto}\rangle \to_e \langle v, \mathit{sto}''\rangle \quad \langle S, \mathit{env}, \mathit{sto}''\rangle \to_s \langle \mathit{env}', \mathit{sto}'\rangle}{\langle e, \mathit{env}, \mathit{sto}\rangle \to_s \langle \epsilon, \mathit{env}', \mathit{sto}'\rangle}$$

[IfStmt-1]

$$\begin{array}{l} \mathit{env} \vdash \langle e, \mathit{sto}\rangle \to_e \langle v', \mathit{sto}''\rangle \quad \langle S_1, \mathit{env}, \mathit{sto}''\rangle \to_s \langle v'', \mathit{env}'', \mathit{sto}^{(3)}\rangle \\ \dfrac{\quad\langle S_2, \mathit{env}'', \mathit{sto}^{(3)}\rangle \to_s \langle v, \mathit{env}', \mathit{sto}'\rangle}{\langle \text{if } e:\ S_1; S_2, \mathit{env}, \mathit{sto}\rangle \to_s \langle v, \mathit{env}, \mathit{sto}'\rangle} \end{array}$$

$$\text{if } v' = True \text{ and } v'' = \epsilon$$

[IfStmt-2]

$$\frac{\mathit{env} \vdash \langle e, \mathit{sto}\rangle \to_e \langle v', \mathit{sto}''\rangle \quad \langle S_1, \mathit{env}, \mathit{sto}''\rangle \to_s \langle v, \mathit{env}', \mathit{sto}'\rangle}{\langle \text{if } e:\ S_1; S_2, \mathit{env}, \mathit{sto}\rangle \to_s \langle v, \mathit{env}, \mathit{sto}'\rangle}$$

$$\text{if } v' = True \text{ and } v \neq \epsilon$$

[IfStmt-3]

$$\frac{\mathit{env} \vdash \langle e, \mathit{sto}\rangle \to_e \langle v', \mathit{sto}''\rangle \quad \langle S_2, \mathit{env}, \mathit{sto}''\rangle \to_s \langle v, \mathit{env}', \mathit{sto}'\rangle}{\langle \text{if } e:\ S_1; S_2, \mathit{env}, \mathit{sto}\rangle \to_s \langle v, \mathit{env}, \mathit{sto}'\rangle}$$

$$\text{if } v' = False$$

Table 5.6: Big-step Semantics for Statements, Continued

[IfElseStmt-1]

$$\frac{\begin{array}{c} env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad \langle S_1, env, sto''\rangle \rightarrow_s \langle v'', env'', sto^{(3)}\rangle \\ \langle S_3, env'', sto^{(3)}\rangle \rightarrow_s \langle v, env', sto'\rangle \end{array}}{\langle \text{if } e: \ S_1 \text{ else } S_2; S_3, env, sto\rangle \rightarrow_s \langle v, env, sto'\rangle}$$

if $v' = True$ and $v'' = \epsilon$

[IfElseStmt-2]

$$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad \langle S_1, env, sto''\rangle \rightarrow_s \langle v, env', sto'\rangle}{\langle \text{if } e: \ S_1 \text{ else } S_2; S_3, env, sto\rangle \rightarrow_s \langle v, env, sto'\rangle}$$

if $v' = True$ and $v \neq \epsilon$

[IfElseStmt-3]

$$\frac{\begin{array}{c} env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad \langle S_2, env, sto''\rangle \rightarrow_s \langle v'', env'', sto^{(3)}\rangle \\ \langle S_3, env'', sto^{(3)}\rangle \rightarrow_s \langle v, env', sto'\rangle \end{array}}{\langle \text{if } e: \ S_1 \text{ else } S_2; S_3, env, sto\rangle \rightarrow_s \langle v, env, sto'\rangle}$$

if $v' = False$ and $v'' = \epsilon$

[IfElseStmt-4]

$$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad \langle S_2, env, sto''\rangle \rightarrow_s \langle v, env', sto'\rangle}{\langle \text{if } e: \ S_1 \text{ else } S_2; S_3, env, sto\rangle \rightarrow_s \langle v, env, sto'\rangle}$$

if $v' = False$ and $v \neq \epsilon$

[WhileStmt-1]

$$\frac{\begin{array}{c} env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad \langle S_1, env, sto''\rangle \rightarrow_s \langle v'', env', sto^{(3)}\rangle \\ \langle \text{while } e: \ S_1; S_2, env, sto^{(3)}\rangle \rightarrow_s \langle v, env'', sto'\rangle \end{array}}{\langle \text{while } e: \ S_1; S_2, env, sto\rangle \rightarrow_s \langle v, env, sto'\rangle}$$

if $v' = True$ and $v'' = \epsilon$

[WhileStmt-2]

$$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad \langle S_1, env, sto''\rangle \rightarrow_s \langle v, env', sto'\rangle}{\langle \text{while } e: \ S_1; S_2, env, sto\rangle \rightarrow_s \langle v, env, sto'\rangle}$$

if $v' = True$ and $v \neq \epsilon$

Table 5.7: Big-step Semantics for Statements, Continued

[WhileStmt-3] $$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad \langle S_2, env, sto''\rangle \rightarrow_s \langle v, env', sto'\rangle}{\langle \text{while } e:\ S_1; S_2, env, sto\rangle \rightarrow_s \langle v, env, sto'\rangle}$$

if $v' = False$

[ReturnStmt] $$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v, sto'\rangle}{\langle \text{return } e, env, sto\rangle \rightarrow_s \langle v, env, sto'\rangle}$$

[EmptyStmt] $\langle \epsilon, env, sto\rangle \rightarrow_s \langle \epsilon, env, sto\rangle$

Table 5.8: Big-step Semantics for Declaration of Variables

[VarDecl-1]
$$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v, sto''\rangle \quad \langle d, env'', sto^{(3)}\rangle \rightarrow_d \langle env', sto'\rangle}{\langle id{=}e, d, env, sto\rangle \rightarrow_d \langle env', sto'\rangle}$$

Where $l = sto''(next)$
$env'' = env[id \mapsto l]$
$sto^{(3)} = sto''[l \mapsto v][next \mapsto new(l)]$

[VarDecl-2]
$$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v, sto''\rangle}{\langle id{=}e, env, sto\rangle \rightarrow_d \langle env', sto'\rangle}$$

Where $l = sto''(next)$
$env' = env[id \mapsto l]$
$sto' = sto''[l \mapsto v][next \mapsto new(l)]$

[VarDecl-3]
$$\frac{\langle d, env'', sto''\rangle \rightarrow_d \langle env', sto'\rangle}{\langle id,\ d, env, sto\rangle \rightarrow_d \langle env', sto'\rangle}$$

Where $l = sto(next)$
$env'' = env[id \mapsto l]$
$sto'' = sto[next \mapsto new(l)]$

[VarDecl-4] $\langle id, env, sto\rangle \rightarrow_d \langle env', sto'\rangle$

Where $l = sto(next)$
$env' = env[id \mapsto l]$
$sto' = sto[next \mapsto new(l)]$

Table 5.9: Big-step Semantics for Expressions

[BinaryExpr]

$$\frac{env \vdash \langle e_1, sto\rangle \rightarrow_e \langle v''sto''\rangle \quad env \vdash \langle e_2, sto''\rangle \rightarrow_e \langle v'sto'\rangle}{env \vdash \langle e_1 \text{ op } e_2, sto\rangle \rightarrow_e \langle v, sto'\rangle}$$

where $v = Apply(op, v'', v')$

[UnaryExpr]

$$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto'\rangle}{env \vdash \langle \text{op } e, sto\rangle \rightarrow_e \langle v, sto'\rangle}$$

where $v = Apply(op, v')$

[ConstructExpr]

$$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad env'' \vdash \langle members, env'', sto''\rangle \rightarrow_m \langle env', sto'\rangle}{env \vdash \langle e(\ ), sto\rangle \rightarrow_e \langle v, sto'\rangle}$$

If $v' = (members, env'')$ and $v = env'$

[CallExpr]

$$\frac{\begin{array}{l} env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto''\rangle \quad env \vdash \langle param, arg, env'', sto''\rangle \rightarrow_{pa} \langle env^{(3)}, sto^{(3)}\rangle \\ \langle S, env^{(3)}, sto^{(3)}\rangle \rightarrow_s \langle v, env', sto'\rangle \end{array}}{env \vdash \langle e(arg), sto\rangle \rightarrow_e \langle v, sto'\rangle}$$

If $v' = (S, param, env'')$

[VarAssignExpr]

$$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v, sto''\rangle \quad env \vdash \langle ve, sto''\rangle \rightarrow_{ve} \langle l, sto^{(3)}\rangle}{env \vdash \langle ve{=}e, sto\rangle \rightarrow_e \langle v, sto^{(3)}[l \mapsto v]\rangle}$$

[VarExpr]

$$\frac{env \vdash \langle ve, sto\rangle \rightarrow_{ve} \langle l, sto'\rangle}{env \vdash \langle ve, sto\rangle \rightarrow_e \langle v, sto'\rangle}$$

Where $v = sto'(l)$

Table 5.10: Big-step Semantics for Expressions, Continued

[ParenExpr] $$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v, sto'\rangle}{env \vdash \langle (e), sto\rangle \rightarrow_e \langle v, sto'\rangle}$$

[AnonymExpr] $env \vdash \langle (param)\text{-}\!> \ type : S, sto\rangle \rightarrow_e \langle v, sto\rangle$

where $v = (S, param, env)$

[NumExpr] $env \vdash \langle n, sto\rangle \rightarrow_e \langle v, sto'\rangle$

where $v = \mathcal{N}[\![v]\!]$

[TrueLiteral] $env \vdash \langle \mathbf{True}, sto\rangle \rightarrow_e \langle True, sto\rangle$

[FalseLiteral] $env \vdash \langle \mathbf{False}, sto\rangle \rightarrow_e \langle False, sto\rangle$

Table 5.11: Big-step Semantics for Variable Expressions

[VarAttAccess] $$\frac{env \vdash \langle e, sto\rangle \rightarrow_e \langle v', sto'\rangle}{\langle e.id, sto\rangle \rightarrow_{ve} \langle l, sto'\rangle}$$

Where $v' = env'$ and $l = env'(id)$

[VarAccess] $env \vdash \langle id, sto\rangle \rightarrow_{ve} \langle l, sto\rangle$

Where $l = env(id)$

Table 5.12: Big-step Semantics for Labels

[UnassignedLabel] $$\frac{v'' \vdash \langle label, env'', sto'' \rangle \rightarrow_l \langle env', sto' \rangle}{v \vdash \langle id; label, env, sto \rangle \rightarrow_l \langle env', sto' \rangle}$$

where $v'' = v + 1$
$l = sto(next)$
$env'' = env[id \mapsto l]$
$sto'' = sto[l \mapsto v][next \mapsto new(l)]$

[Label] $$\frac{v^{(3)} \vdash \langle label, env'', sto'' \rangle \rightarrow_l \langle env', sto' \rangle}{v \vdash \langle id = n; label, env, sto \rangle \rightarrow_l \langle env', sto' \rangle}$$

where $\mathcal{N}[\![n]\!] = v''$ and $v'' \geq v$
$v^{(3)} = v'' + 1$
$l = sto(next)$
$env'' = env[id \mapsto l]$
$sto'' = sto[l \mapsto v''][next \mapsto new(l)]$

[EmptyLabel] $$v \vdash \langle \epsilon, env, sto \rangle \rightarrow_l \langle env, sto \rangle$$

Table 5.13: Big-step Semantics for Arguments and Parameters

[SeqArgParam]

$$\frac{env^{(3)} \vdash \langle e, sto\rangle \rightarrow_e \langle v, sto''\rangle \quad env^{(3)} \vdash \langle param, arg, env'', sto^{(3)}\rangle \rightarrow_{pa} \langle env', sto'\rangle}{env^{(3)} \vdash \langle id_t\ id, param, e{,}arg, env, sto\rangle \rightarrow_{pa} \langle env', sto'\rangle}$$

Where $l = sto''(next)$
$env'' = env[id \mapsto l]$
$sto^{(3)} = sto''[l \mapsto v][next \mapsto new(l)]$

[SeqArgParam]

$$\frac{env'' \vdash \langle e, sto\rangle \rightarrow_e \langle v, sto''\rangle}{env'' \vdash \langle id_t\ id, e, env, sto\rangle \rightarrow_{pa} \langle env', sto'\rangle}$$

Where $l = sto''(next)$
$env' = env[id \mapsto l]$
$sto' = sto''[l \mapsto v][next \mapsto new(l)]$

[EmptyArgParam] $env' \vdash \langle \epsilon, \epsilon, env, sto\rangle \rightarrow_{pa} \langle env, sto\rangle$

CHAPTER 6

Groo Type System

In this chapter we will present the static semantics for groo, or rather, the type system. The purpose of this is to formalise the rules for deciding if a groo program is type correct.

6.1 Definition of Types

To denote types, we introduce a new syntactic category called **Types**. Elements in **Types** will be denoted by the metavariable T. A subset of types is the primitive types, **Primitives**. Elements of **Primitives** are denoted by the metavariable $Prim$.

The productions for **Types** are listed in table 6.1. $(T_1 \rightarrow T_2)$ denotes a function type with T_1 as the type of the arguments, and T_2 as the return type. $T_1 \times T_2$ denotes a tuple of types, this is used to create a function type with multiple parameters. $(id, members)$ denotes the type that a class with the name id and body $members$ declares. $(id, labels)$ denotes the type that an enum with the name id and body $labels$ declares.

$$\begin{array}{lll} Prim & ::= & int \mid float \mid bool \mid void \\ T & ::= & Prim \mid T_1 \times T_2 \mid (T_1 \rightarrow T_2) \mid (id, members) \mid (id, labels) \end{array}$$

Table 6.1: Abstract syntax for types.

6.2 Variable Type

Below we have defined the set **VariableType**, which used in our environments to denote whether a declared variable is read-only. A element of **VariableType** is denoted by the metavariable vt. Elements of the auxiliary set $\mathcal{P}(\{Read, Write\})$ are denoted by the metavariable R. We use this variable type as this enables us to treat methods and constructors as read-only function variables. This emphasises the concept of how methods are closures in groo.

$$\begin{array}{rcl} R & \in & \mathcal{P}(\{Read, Write\}) \\ vt & \in & \textbf{VariableType} = \mathcal{P}(\{Read, Write\}) \times \textbf{Types} \end{array}$$

6.3 Environments

In groo, an id can either be a variable and a type. For example, this applies to classes where the name of the class is both the class type and the constructor type. This results in a nominal type system. Therefore we introduce two environments:

- a type declaration environment to hold all type declarations,
- and a variable declaration environment to hold variables and their types.

The type declaration environment is defined as:

$$dt \in DT = \mathbf{ID} \rightharpoonup \mathbf{Types}$$

and the variable declaration environment as:

$$dv \in DV = \mathbf{ID} \cup \{T_{ret}, impRet\} \rightharpoonup \mathcal{P}(\{Read, Write\}) \times \mathbf{Types} \cup \mathbf{Types} \cup \{True, False\}$$

Where T_{ret} and $impRet$ are special elements. T_{ret} is used to find the return type and $impRet$ is used to determine if an empty statement constitutes an implicit void return. E.g. composite statements' $impRet$ is set to $false$ in dv for the first statement, and $true$ in the last statement. The result of this is that $impRet$ only is true for the last statement in a scope. Likewise, this is done for other statements which have more than one statement as their immediate constituents.

Another approach to this could be to let statements return a type and then compare this in call expressions and alike. However, implicit return at different scopes makes it complicated, and to remove $impRet$ would double the number of rules. Because ϵ at the outer most scope of a function means an implicit void return, and in every other scope in that function, ϵ means no return. ϵ is therefore treated differently at different scopes. Removing T_{ret} would complicate the side conditions of composite statements such as if and while unnecessarily. Thus we have chosen to use the slightly awkward special elements T_{ret} and $impRet$.

6.3.1 Standard Environment

The standard environment, $std \in DT$, contains the primitive types. It is defined as:

$$\begin{aligned} std(\mathbf{int}) &= Int \\ std(\mathbf{float}) &= Float \\ std(\mathbf{bool}) &= Bool \\ std(\mathbf{void}) &= Void \end{aligned}$$

The standard environment is used as a type declaration, dt, in the type rule for a groo program.

6.4 Auxiliary Functions

A number of auxiliary functions that are needed for the type judgements of groo.

6.5 $Apply_T$ Function

The $Apply_T$ function takes an operator, one or two types and returns a type. The $Apply_T$ function is defined as in table 6.2.

Table 6.2: $Apply_T$ function

$$Apply_T(+, Int, Int) = Int \tag{6.1}$$
$$Apply_T(+, Float, Float) = Float \tag{6.2}$$
$$Apply_T(+, String, String) = String \tag{6.3}$$
$$Apply_T(-, Int, Int) = Int \tag{6.4}$$
$$Apply_T(-, Float, Float) = Float \tag{6.5}$$
$$Apply_T(*, Int, Int) = Int \tag{6.6}$$
$$Apply_T(*, Float, Float) = Float \tag{6.7}$$
$$Apply_T(/, Int, Int) = Int \tag{6.8}$$
$$Apply_T(/, Float, Float) = Float \tag{6.9}$$
$$Apply_T(<<, Int, Int) = Int \tag{6.10}$$
$$Apply_T(>>, Int, Int) = Int \tag{6.11}$$
$$Apply_T(\%, Int, Int) = Int \tag{6.12}$$
$$Apply_T(||, Bool, Bool) = Bool \tag{6.13}$$
$$Apply_T(\mathbf{or}, Bool, Bool) = Bool \tag{6.14}$$
$$Apply_T(\&\&, Bool, Bool) = Bool \tag{6.15}$$
$$Apply_T(\mathbf{and}, Bool, Bool) = Bool \tag{6.16}$$
$$Apply_T(==, T, T) = Bool \tag{6.17}$$
$$Apply_T(! =, T, T) = Bool \tag{6.18}$$
$$Apply_T(<, Int, Int) = Int \tag{6.19}$$
$$Apply_T(<, Float, Float) = Float \tag{6.20}$$
$$Apply_T(< =, Int, Int) = Int \tag{6.21}$$
$$Apply_T(< =, Float, Float) = Float \tag{6.22}$$
$$Apply_T(>, Int, Int) = Int \tag{6.23}$$
$$Apply_T(>, Float, Float) = Float \tag{6.24}$$
$$Apply_T(> =, Float, Float) = Float \tag{6.25}$$
$$Apply_T(> =, Int, Int) = Int \tag{6.26}$$
$$Apply_T(-, Int) = Int \tag{6.27}$$
$$Apply_T(-, Float) = Float \tag{6.28}$$
$$Apply_T(!, Bool) = Bool \tag{6.29}$$

6.5.1 The Set Function

$set(T)$ is a function used to check whether a value is of type Int or Float. The function is defined as:

$$\begin{aligned} set(Int) &= \mathbb{Z} \\ set(Float) &= \mathbb{Q} \end{aligned}$$

6.5.2 Domain of Partial Functions

$D(f)$ denotes the domain of the partial function f, i.e. the domain of a $dt \in DT$ is the set of ids declared by the type declaration environment, dt. This is used to ensure that when we declare a new variable an id of the same name is not already declared.

6.5.3 Update Function for Type Declaration Environments

Updating a type declaration environment is denoted $dt[id \mapsto T]$ and the resulting type declaration environment is defined as:

$$dt'(x) = \begin{cases} dt(x) & \text{if } id \neq x \\ T & \text{if } id = x \end{cases}$$

6.5.4 Update Function for Variable Declaration Environments

Updating a variable declaration environment is denoted $dv[id \mapsto vt]$ and the resulting variable declaration environment is defined as:

$$dv'(x) = \begin{cases} dv(x) & \text{if } id \neq x \\ vt & \text{if } id = x \end{cases}$$

6.5.5 dt from Type Declarations

To make a dt (type declaration environment) from a set of type declarations the following partial auxiliary function is introduced.

$$\begin{aligned} dt_d(\epsilon, dt) &= dt \\ dt_d(\textbf{class}\ id : members\ ; decls, dt) &= dt_d(decls, dt[id \mapsto (id, members)]) \\ dt_d(\textbf{enum}\ id : labels\ ; decls, dt) &= dt_d(decls, dt[id \mapsto (id, labels)]) \end{aligned}$$

We recursively call this function continuously updating dt with the id pointing to a type declaration, creating a new type. The type of a class is a tuple of its name and its members, as we have a nominal type system.

6.5.6 dv from Type Declarations

To make a dv from type declarations the following partial auxiliary function is introduced. This function is used to make constructor types.

$$\begin{aligned}
dv_t(\epsilon, dv) &= dv \\
dv_t(\textbf{class}\ id : members\ ;\ decls) &= dv_t(decls, dv[id \mapsto (\{Read\}, (Void \rightarrow (id, members)))]) \\
dv_t(\textbf{enum}\ id : labels\ ;\ decls) &= dv_t(decls, dv[id \mapsto (\{Read\}, (Void \rightarrow (id, labels)))])
\end{aligned}$$

The declarations of a class implicitly creates a read-only variable which is the constructor of the class. Therefore it has the type from: void to the type of the class.

6.5.7 dv From Variable Declarations

To make a dv from variable declarations the following partial auxiliary function is introduced. Note that this function is not defined for the given parameters if the side condition is not ok or evaluates to a type, for statement and expression side conditions respectively.

$$\begin{aligned}
dv_d(id = e, S, dt, dv) &= dv[id \mapsto (\{Read, Write\}, T)] \quad \text{, where } dt, dv \vdash \langle e \rangle : T \\
dv_d(id = e, d, S, dt, dv) &= dv_d(d, S, dt, dv[id \mapsto (\{Read, Write\}, T)]), \text{where } dt, dv \vdash \langle e \rangle : T \\
dv_d(id,\ S, dt, dv) &= dv[id \mapsto (\{Read, Write\}, T)] \\
&\quad\ \text{, where } dt, dv[id \mapsto T] \vdash \langle S \rangle : ok \\
dv_d(id,\ d, S, dt, dv) &= dv_d(d, S, dt, dv[id \mapsto (\{Read, Write\}, T)]) \\
&\quad\ \text{, where } dt, dv[id \mapsto T] \vdash \langle S \rangle : ok
\end{aligned}$$

This auxiliary function is used to give types to variables when they are declared. If the variable is assigned an expression, the id will get the type of the expression. If it is not assigned any expression, we give it a type such that the subsequent statements will be well typed.

6.5.8 Tp From Parameter Declarations

To make a Tp from parameter declarations the following partial auxiliary function is introduced.

$$\begin{aligned}
Tp_p(type\ id, param, dt) &= Tp_p(param, T_t(type, dt), dt) \\
Tp_p(type\ id\ , param, T, dt) &= Tp_p(param, T \times T_t(type, dt), dt) \\
Tp_p(type\ id, T, dt) &= T \times T_t(type, dt) \\
Tp_p(\epsilon, dt) &= Void \\
Tp_p(type\ id, dt) &= T_t(type, dt)
\end{aligned}$$

The first is used to begin a tuple, where the function calls itself with the remaining parameters and a tuple of the type of the first parameter. The second adds the type of parameter to the given tuple, and calls the function again. The third adds the type of the last parameter to the tuple and returns it.

6.5.9 dv From Parameter Declarations

To make a dv from parameter declarations the following partial auxiliary function is introduced.

$$\begin{aligned}
dv_p(type\ id, dt, dv) &= dv[id \mapsto (\{Read, Write\}, T_t(type, dt))] \\
dv_p(type\ id\ , param, dt, dv) &= dv_p(param, dt, dv[id \mapsto (\{Read, Write\}, T_t(type, dt))])
\end{aligned}$$

Each parameter is assigned a type in dv.

6.5.10 dv From Label Declarations

To make a dv from label declarations the following partial auxiliary function is introduced.

$$\begin{aligned}
dv_l(id, labels, dt, dv) &= dv[id \mapsto (\{Read\}, dt(Int))] \\
dv_l(id = n,\ labels, dt, dv) &= dv_p(labels, dt, dv[id \mapsto (\{Read\}, dt(Int))])
\end{aligned}$$

Each labels is assigned a integer type in dv.

6.5.11 dv From Member Declarations

To make a dv from member declarations the following partial auxiliary function is introduced.

$$\begin{aligned}
dv_m(\epsilon, dt, dv) &= dv \\
dv_m(type\ id;\ members, dt, dv) &= dv_m(members, dt, dv[id \mapsto vt]) \\
&\quad ,\ \text{where } vt = (\{Read, Write\}, T_t(type, dt)) \\
dv_m(type\ id = e;\ members, dt, dv) &= dv_m(members, dt, dv[id \mapsto vt]) \\
&\quad ,\ \text{where } vt = (\{Read, Write\}, T_t(type, dt)) \\
dv_m(type\ id\ (\ param\ ) : S\ ; members, dt, dv) &= dv_m(members, dt, dv[id \mapsto (\{Read\}, T)]) \\
&\quad ,\ \text{where } T = (Tp_p(param, dt) \rightarrow T_t(type, dt))
\end{aligned}$$

This auxiliary function is used to give types to members. Fields are given the type declared, and methods are given a tuple of the types of the parameters and the methods return type.

6.5.12 Tp From Arguments

To make a Tp from arguments the following auxiliary partial function is introduced. This function is only defined for these parameters and if the side conditions are ok or evaluates to a type for the statement or expression side conditions respectively.

$$\begin{array}{rll}
Tp_a(e, dt, dv, T_1) & = T_1 \times T_2 & \text{where } dt, dv \vdash \langle e \rangle : T_2 \\
Tp_a(e, dt, dv) & = T & \text{where } dt, dv \vdash \langle e \rangle : T \\
Tp_a(\epsilon, dt, dv) & = Void & \\
Tp_a(e\ ,\ arg, dt, dv) & = Tp_a(arg, dt, dv, T) & \text{where } dt, dv \vdash \langle e \rangle : T \\
Tp_a(e\ ,\ arg, dt, dv, T_1) & = Tp_a(arg, dt, dv, T_1 \times T_2) & \text{where } dt, dv \vdash \langle e \rangle : T_2
\end{array}$$

Creating a tuple from arguments is somewhat like as from parameters, the only difference is that the type is the type of the expression.

6.5.13 T From Type Annotations

To make a type,T, from type annotations the following partial auxiliary function is introduced.

$$\begin{array}{rcl}
T_t(id, dt) & = & dt(id) \\
T_t(id, dt, Tp) & = & tp \times dt(id) \\
T_t(id, type, dt) & = & T_t(type, dt, dt(id)) \\
T_t(id, type, dt, Tp) & = & T_t(type, dt, Tp \times dt(id)) \\
T_t(type_1\text{->}type_2, dt) & = & (T_t(type_1, dt) \rightarrow T_t(type_2, dt))
\end{array}$$

6.6 Type Judgements

In this section we will list the type judgements for groo. The type of expressions and variable expressions are $T \in Types$, and are well typed, ok, if they comply with the type rules.

groo Program

The type judgement

$$\langle groo \rangle : ok$$

means that the program is a well typed groo program. A program is a groo program if its declarations are well typed. The type rule can be seen in table 6.3.

Declarations

The type judgement

$$dt, dv \vdash \langle decls \rangle : ok$$

means that given the variable declarations, dv, and the type declarations, dt, the declaration $decls$ is well typed. A declaration is well typed if its members and are well typed and that the type has not been overwritten by an other declaration in the auxiliary function. The type rules can be seen in table 6.4 where we use the above mentioned auxiliary function to declare the type of the declarations.

Members

The type judgement

$$dt, dv \vdash \langle members\rangle : ok$$

means that given the type declarations, dt, and the variable declarations, dv, the member is well typed. Field members are well typed if their declared type and the type of the assigned expression - if any - are both equal to the type associated with the id of the field xin dv.

Method members are well typed if their body and parameters are well typed and that their types are declared in dt.

The type rules for members can be seen in table 6.5.

Statements

The type judgement

$$dt, dv \vdash \langle S\rangle : ok$$

means that given type declarations, dt, and variable declarations, dv, the statement S is type correct. The type rules for statements can be seen in table 6.6. For statements with conditions, the conditional expression must have a boolean type. In the return statement, we check that the expression has a type, and compares this type to the element T_{ret} in dv maps to. An empty statement is well typed if the return type has been set to $Void$ or there is no implicit return.

Variable Declarations

The type judgement

$$dt, dv \vdash \langle d\rangle : ok$$

means that given type declarations, dt, and variable declarations dv, the variable declaration d is well typed. A variable declaration is well typed if the expressions have a type, and if the id is not declared before, i.e $id \notin D(dv)$. The type rules can be seen in table 6.7.

Expressions

The type judgement for expressions

$$dt, dv \vdash \langle e\rangle : T$$

means that given type declarations dt and variable declarations dv the expression e has type T. The type rules given in table 6.8 and 6.9.

Variable Expressions

The type judgement for variable expressions

$$dt, dv \vdash \langle ve \rangle : vt$$

means that given the type declarations, dt, and the variable declartions, dv, the variable expression, ve, has variable type vt indicating whether it is read, write or read-write and the type. This type is found by looking up id in dv. Table 6.10 lists the type rules.

Arguments

The type judgement for arguments

$$dt, dv \vdash \langle arg \rangle : ok$$

means that given type declarations, dt, and variable declarations, dv, the argument is well typed. An argument is well typed if the expression has a type. The type rules can be seen in table 6.11

Parameters

The type judgement for parameters

$$dt, dv \vdash \langle param \rangle : ok$$

means that given type declarations, dt, and variable declarations, dv, the parameter is well typed. A parameter is well typed if the declared type is in dt and $id \notin D(dv)$. That is, the type is declared whereas the variable is not. The type rules can be seen in table 6.12.

Labels

The type judgement for labels

$$dt, dv \vdash \langle labels \rangle : ok$$

means that given type declarations, dt, and variable declarations, dv, the label is well typed. A label is well typed if it is assigned an integer.

Table 6.3: Type judgements for program, groo

[groo] $$\frac{dt', dv' \vdash \langle decls \rangle : ok}{\langle decls \rangle : ok}$$

where $dt' = dt_d(decls, std)$ and $dv' = dv_t(decls, \{\})$

Table 6.4: Type judgements for declarations, decls

[ClassDecl] $$\frac{dt, dv' \vdash \langle members \rangle : ok \quad dt, dv \vdash \langle decls \rangle : ok}{dt, dv \vdash \langle \textbf{class}\ id\text{: } members; decls \rangle : ok}$$

where $dv' = dv_m(members, dt, dv), dt(id) = (id, members)$
and $dv(id) = (\{Read\}, (Void \rightarrow (id, members)))$

[EnumDecl] $$\frac{dt, dv \vdash \langle labels \rangle : ok \quad dt, dv \vdash \langle decls \rangle : ok}{dt, dv \vdash \langle \textbf{enum}\ id\text{: } labels; decls \rangle : ok}$$

where $dt(id) = (id \rightarrow labels)$
and $dv(id) = (\{Read\}, (Void \rightarrow (id, labels)))$

[EmptyDecl] $dt, dv \vdash \langle \epsilon \rangle : ok$

Table 6.5: Type judgements for members, members

[FieldMember]
$$\frac{dt, dv \vdash \langle members \rangle : ok}{dt, dv \vdash \langle type\ id; members \rangle : ok}$$

if $dv(id) = (\{Read, Write\}, T_t(type, dt))$

[FieldMemberExt]
$$\frac{dt, dv \vdash \langle e \rangle : T \quad dt, dv \vdash \langle members \rangle : ok}{dt, dv \vdash \langle type\ id = e; members \rangle : ok}$$

if $dv(id) = (\{Read, Write\}, T_t(type, dt))$ and $T_t(type, dt) = T$

[MethodMember]
$$\frac{dt, dv \vdash \langle param \rangle : ok \quad dt, dv' \vdash \langle S \rangle : ok \quad dt, dv \vdash \langle members \rangle : ok}{dt, dv \vdash \langle type\ id(param) : S; members \rangle : ok}$$

where $dv' = dv_p(param, dt, dv)[T_{ret} \mapsto T_t(type, dt)][impRet \mapsto True]$
and $dv(id) = (\{Read\}, (Tp_p(param, dt) \rightarrow T_t(type, dt)))$

[EmptyMember] $dt, dv \vdash \langle \epsilon \rangle : ok$

Table 6.6: Type judgements for statements, Stmt

[DeclStmt]
$$\frac{dt, dv \vdash \langle d \rangle : ok \quad dt, dv' \vdash \langle S \rangle : ok}{dt, dv \vdash \langle \mathbf{var}\ d; S \rangle : ok}$$

where $dv' = dv_d(d, dt, dv)$

[ExprStmt]
$$\frac{dt, dv \vdash \langle e \rangle : T \quad dt, dv \vdash \langle S \rangle : ok}{dt, dv \vdash \langle e; S \rangle : ok}$$

[IfStmt]
$$\frac{dt, dv \vdash \langle e \rangle : Bool \quad dt, dv' \vdash \langle S_1 \rangle : ok \quad dt, dv \vdash \langle S_2 \rangle : ok}{dt, dv \vdash \langle \mathbf{if}\ e : S_1; S_2 \rangle : ok}$$

where $dv' = dv[impRet \mapsto False]$

[IfElseStmt]
$$\frac{dt, dv \vdash \langle e \rangle : Bool \quad dt, dv' \vdash \langle S_1 \rangle : ok \quad dt, dv' \vdash \langle S_2 \rangle : ok \quad dt, dv \vdash \langle S_3 \rangle : ok}{dt, dv \vdash \langle \mathbf{if}\ e : S_1\ \mathbf{else}\ S_2; S_3 \rangle : ok}$$

where $dv' = dv[impRet \mapsto False]$

[WhileStmt]
$$\frac{dt, dv \vdash \langle e \rangle : Bool \quad dt, dv' \vdash \langle S_1 \rangle : ok \quad dt, dv \vdash \langle S_2 \rangle : ok}{dt, dv \vdash \langle \mathbf{while}\ e : S_1; S_2 \rangle : ok}$$

where $dv' = dv[impRet \mapsto False]$

[ReturnStmt]
$$\frac{dt, dv \vdash \langle e \rangle : T}{dt, dv \vdash \langle \mathbf{return}\ e \rangle : ok}$$

where $dv(T_{ret}) = T$

[EmptyStmt] $dt, dv \vdash \langle \epsilon \rangle : ok$

where either $dv(T_{ret}) = Void$ or $dv(impRet) = False$

Table 6.7: Type judgements for variable declarations, VarDecl

[VarDecl-1] $$\frac{dt, dv \vdash \langle e \rangle : T \quad dt, dv[id \mapsto (\{Read, Write\}, T)] \vdash \langle d \rangle : ok}{dt, dv \vdash \langle id = e, d \rangle : ok}$$

where $id \notin D(dv)$

[VarDecl-2] $$\frac{dt, dv \vdash \langle e \rangle : T}{dt, dv \vdash \langle id = e \rangle : ok}$$

where $id \notin D(dv)$

[VarDecl-3] $$\frac{dt, dv[id \mapsto (\{Read, Write\}, Void)] \vdash \langle d \rangle : ok}{dt, dv \vdash \langle id, d \rangle : ok}$$

where $id \notin D(dv)$

[VarDecl-4] $$dt, dv \vdash \langle id \rangle : ok$$

where $id \notin D(dv)$

Table 6.8: Type judgements for expressions, Expr

[BinaryExpr] $$\frac{dt, dv \vdash \langle e_1 \rangle : T' \quad dt, dv \vdash \langle e_2 \rangle : T''}{dt, dv \vdash \langle e_1 \; op \; e_2 \rangle : T}$$

where $T = Apply_T(op, T', T'')$

[UnaryExpr] $$\frac{dt, dv \vdash \langle e \rangle : T'}{dt, dv \vdash \langle op \; e \rangle : T}$$

where $T = Apply_T(op, T')$

[VarExpr] $$\frac{dt, dv \vdash \langle ve \rangle : vt}{dt, dv \vdash \langle ve \rangle : T}$$

where $vt = (R, T)$ and $Read \in R$

[AssignExpr] $$\frac{dt, dv \vdash \langle ve \rangle : vt \quad dt, dv \vdash \langle e \rangle : T}{dt, dv \vdash \langle ve = e \rangle : T}$$

where $vt = (R, T)$ and $Write \in R$

[CallExpr] $$\frac{dt, dv \vdash \langle e \rangle : T' \quad dt, dv \vdash \langle arg \rangle : ok}{dt, dv \vdash \langle e(arg) \rangle : T}$$

where $T' = (Tp \to T)$ and $Tp_a(arg, dt, dv) = Tp$

[AnonymExpr] $$\frac{dt, dv \vdash \langle param \rangle : ok \quad dt, dv' \vdash \langle S \rangle : ok}{dt, dv \vdash \langle (param)\text{->}id : S \rangle : T}$$

where $T = (Tp_p(param), dt, dv) \to dt(id))$
and $dv' = dv_p(param, dt, dv)[T_{ret} \mapsto dt(id)][impRet \mapsto True]$

Table 6.9: Type judgements for expressions, Expr, continued

[ParenExpr] $$\frac{dt, dv \vdash \langle e \rangle : T}{dt, dv \vdash \langle (e) \rangle : T}$$

[NumLiteral-1] $dt, dv \vdash \langle n \rangle : Int$

where $n \in set(Int)$

[NumLiteral-2] $dt, dv \vdash \langle n \rangle : Float$

where $n \in set(Float)$

[TrueLiteral] $dt, dv \vdash \langle True \rangle : Bool$

[FalseLiteral] $dt, dv \vdash \langle False \rangle : Bool$

Table 6.10: Type judgements for variable expressions, VarExpr

[VarAccess] $dt, dv \vdash \langle id \rangle : vt$

where $dv(id) = vt$

[VarAttAccess1] $$\frac{dt, dv \vdash \langle e \rangle : T}{dt, dv \vdash \langle e.id \rangle : vt}$$

where $T = (id, members)$,
$dv' = dv_m(members, dt, dv)$ and
$dv'(id) = vt$

[VarAttAccess2] $$\frac{dt, dv \vdash \langle e \rangle : T}{dt, dv \vdash \langle e.id \rangle : vt}$$

where $T = (id, labels)$,
$dv' = dv_l(labels, dt, dv)$ and
$dv'(id) = vt$

Table 6.11: Type judgements for arguments, Arg

[SeqArg] $$\frac{dt, dv \vdash \langle e \rangle : T \quad dt, dv \vdash \langle arg \rangle : ok}{dt, dv \vdash \langle e, arg \rangle : ok}$$

[Arg] $$\frac{dt, dv \vdash \langle e \rangle : T}{dt, dv \vdash \langle e \rangle : ok}$$

[EmptyArg] $dt, dv \vdash \langle \epsilon \rangle : ok$

Table 6.12: Type judgements for parameters, Param

[SeqParam]
$$\frac{dt \vdash \langle type \rangle : ok \quad dt, dv[id \mapsto T_t(type, dt)] \vdash \langle param \rangle : ok}{dt, dv \vdash \langle type\ id, param \rangle : ok}$$

where $id \notin D(dv)$

[Param]
$$\frac{dt \vdash \langle type \rangle : ok}{dt, dv \vdash \langle type\ id \rangle : ok}$$

where $id \notin D(dv)$

[EmptyParam] $dt, dv \vdash \langle \epsilon \rangle : ok$

Table 6.13: Type judgements for labels, Labels

[UnassignedLabel]
$$\frac{dt, dv[id \mapsto dt(Int)] \vdash \langle labels \rangle : ok}{dt, dv \vdash \langle id;\ labels \rangle : ok}$$

where $id \notin D(dv)$

[Label]
$$\frac{dt, dv[id \mapsto dt(Int)] \vdash \langle labels \rangle : ok}{dt, dv \vdash \langle id = n;\ labels \rangle : ok}$$

where $id \notin D(dv)$ and $n \in set(Int)$

[EmptyLabel] $dt, dv \vdash \langle \epsilon \rangle : ok$

Table 6.14: Type judgements for type, Type

[FunctionType] $$\frac{dt \vdash \langle type_1 \rangle : ok \quad dt \vdash \langle type_2 \rangle : ok}{dt \vdash \langle type_1\text{->}type_2 \rangle : ok}$$

[TupleType] $$\frac{dt \vdash \langle type \rangle : ok}{dt \vdash \langle id, type \rangle : ok}$$

where $id \in D(dt)$

[SimpleType] $$dt \vdash \langle id \rangle : ok$$

where $id \in D(dt)$

Part III

Syntax Analysis

CHAPTER 7

Lexing

The purpose of the lexical analysis is to translate the written lexemes into tokens that can be used in the further analysis. A deterministic finite state (DFA) algorithm can be used to do this.

7.1 Deterministic Finite State Automata

A deterministic finite state automaton (DFA) is a simple machine that recognises a regular language. It is a 5-tuple, (Q, Σ, δ, q_0, F), where:

- Q is a finite set of states
- Σ is the alphabet, consisting of a finite set of symbols
- δ is the transition function, ($\delta : Q$ x $\Sigma \rightarrow Q$)
- q_0 is the start state, $q_0 \in \Sigma$
- F is the set of accept states, $F \subseteq Q$

A DFA takes a string over the alphabet Σ as input. The transition function δ takes into account the current state and input symbol and changes the state until the end of the input. If, at the end of the string, the current state is an accept state ($q_{current} \in F$) the DFA accepts the string - that is, the string is an element of the language this DFA recognises.

7.1.1 What are regular expressions?

Regular expressions are a method of describing regular languages.

A regular expression is defined by six clauses, so that R is a regular expression if R is one of the following:

1. a for some symbol in the alphabet Σ
2. ϵ
3. $\emptyset$
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions
6. $(R_1)^*$, where R_1 is a regular expression

Note that R_1 and R_2 are both smaller than R, ensuring that this definition is not circular [Sipser, 2006].

Regular expressions are a concise way of describing patterns. In the context of lexical analysis, regular expressions are used to recognise lexical items. Additionally, regular expressions are relatively easy for humans to write, read, and understand.

In the lexer generator built for constructing groo, regular expressions are used to describe different parts of the language in small blocks. For example, the reserved word *while* has its own regular expression (which is, quite simply, $\{while\}$).

Every regular expression entered into the lexer-generator has an accept rule. When a string that is described by a particular regular expression is entered into the lexer-generator, the output is the accept rule associated with that specific regular expression - as such, a non-accepting state does not have an accept rule. An accept rule contains a terminal, for example the token for the plus operator, and the position. These accept rules are also utilised when minimising the DFA produced (see section 7.4).

7.2 Constructing a DFA

Our lexer generator takes as input a file containing C-commands, which are not processed but merely passed on, and some regular expressions with accept rules.

To ease the writing of the regular expressions, it is possible to create "groups". Instead of having to write the entire alphabet several times, a group can be created at the beginning of the file. A group could be $[a-d]$ $\{a|b|c|d\}$. Now it is possible to write $[a-d]$ in a regular expression, instead of having to write every letter from a to d, and the lexer generator will replace it.

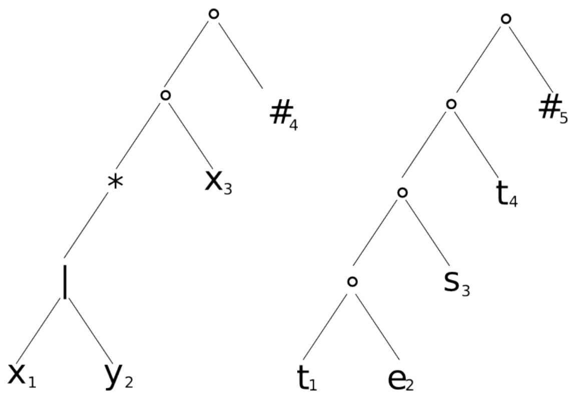
The regular expressions with accept rules are parsed. When the lexer generator has parsed a regular expression and an accept rule, it builds a syntax tree from the regular expression and concatenates it with an AND node with a leaf containing the accept rule.

Concatenating two trees representing regular expressions with an AND node gives the same result as $R_1 \cap R_2$, while concatenation with an OR node has the same result as $R_1 \cup R_2$. A tree can contain kleene nodes, representing the kleene star used in regular expressions ($*$), OR nodes, AND nodes, and leaves containing either symbols or accept rules. If other regular expressions have been entered, and thus other trees have been created before this, they are concatenated using OR, creating a single tree. The lexer generator will then create the DFA from the tree, represented as a table of $goto$s.

Example 1 When processing $\{(x|y)^*x\}$ {return new TOKEN_example} $\{test\}$ {return new TOKEN_test}, the lexer generator will first read "$(x|y)^*x$" as a regular expression and "return new TOKEN_example" as the accept rule.

It creates the tree on figure 7.1a. The leaf containing the accept rule is denoted $\#$, and is in leaf number 4. It then reads the next regular expression, {test} and its accept rule, it creates the tree on figure 7.1b.

Then, the two trees are combined, creating the tree on figure 7.2. From this tree, a DFA is built. The NFA seen on figure 7.3 is a simplified version of the DFA created.



(a) Tree for $\{(x|y)^*x\}$ (b) Tree for $\{test\}$

Figure 7.1: Trees for the regular expressions in example 1

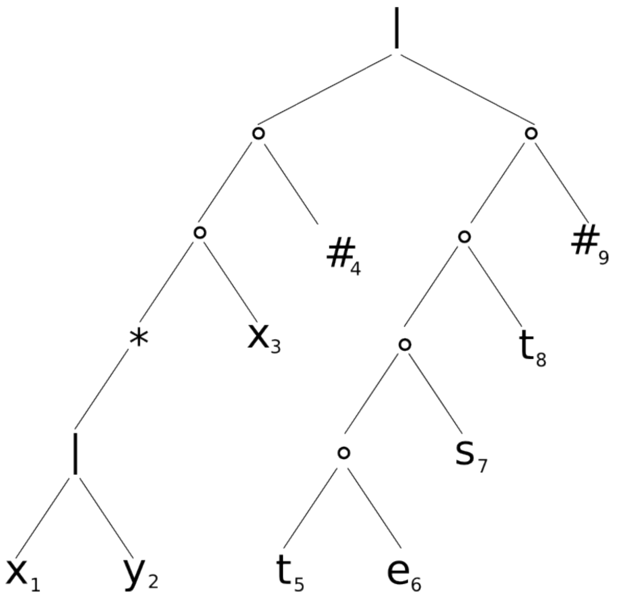


Figure 7.2: The combined tree for $\{(x|y)^*x\}$ and $\{test\}$.

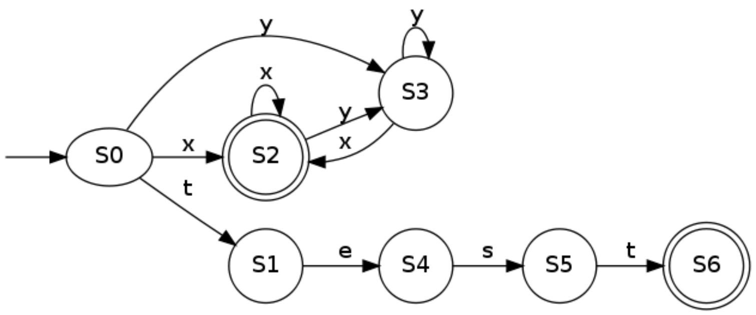


Figure 7.3: NFA for example 1.

The DFA which is generated from the tree has additional transitions from every state, with all other possible input, going to a default state - a sink. As this would create a large and unreadable DFA, we have chosen to only show the relevant transitions.

7.3 Implementing a Lexer

The lexer is implemented in C++ and is heavily inspired by the article by Bumbulis and Cowan [1994].

The DFA is implemented as a collection of states, where each state has a set of transitions, and a label with the state id. The lexer reads a block of source code into a buffer, where a pointer, `limit`, points to the end of the buffer. Every time a state is visited, the `cursor` pointer will be incremented to point to the next symbol of the input. A switch statement will then identify the current symbol, and branch the program to the next state. If the current symbol cannot lead to an accepting state the `default goto` will branch to `final` which is a place in code where a token of the last accepted string will be returned.

```
State1:
    cursor++;
    if (limit == cursor) fill ();
    switch(*cursor){
        case 97: goto State2;  // 'a'
        case 98: goto State3;  // 'b'
        default: goto final ;
    };
```

Listing 7.1: Example State.

If the state is an accepting state additional code is needed.

```

State2:
    cursor++;
    if (limit == cursor) fill ();
    accept = 1;
    marker = cursor;
    switch(*cursor){
        case 49: goto State2;  // '1'
        case 48: goto State3;  // '0'
        default: goto final ;
    };
```

Listing 7.2: Example accept state.

First `accept` will be given a value to identify which accept rule the accept state for this regular expression results in. Second a `marker` will be set to point to the same as the `cursor`. This is used later in the code `final` to retrieve the accepted input.

```
final :
    cursor = marker;
    switch(accept){
        case 1: // user specified action code
            {
                return new StringLiteral(bufstart, marker - bufstart);
            }
            break;
        case 2: // user specified action code
            {
                return new Terminal(TOKEN_comma);
            }
            break;
        default: // user specified default action code
            {
                return new Terminal(TOKEN_error);
            }
            break;
    }
    bufstart = marker;
    goto START;
```

Listing 7.3: Example accept state.

When the DFA reaches a state from which it can no longer reach an accepting state with the current symbol, the lexer will go to `final`. The cursor is set to point to the marker symbol as this is the last symbol in the last accepted string. The lexer can return the accepted string as it is represented by the string in between the `bufstart` and `marker` pointers. When a Token has been returned the `bufstart` pointer is set to `marker` and the lexer is ready to begin reading the next token. The string buffer is illustrated in figure 7.4.

Buffer management

When the `cursor` reaches the `limit` of the buffer the content of the buffer and all its pointers are moved by `bufstart` - `buffer`. The buffer is then filled up from `limit`, and `limit` is moved forward equal to the amount loaded into the buffer. This result in the buffer being refilled, making the lexer ready to continue.

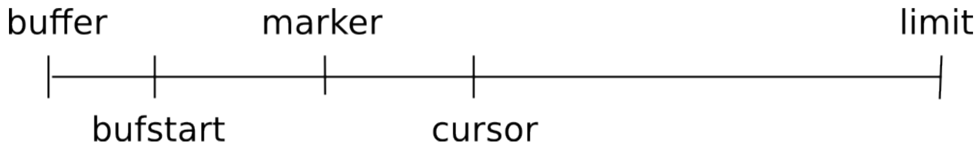


Figure 7.4: The String buffer and its pointers.

Indents

As indents and dedents are represented using white space, groo is not a context free language - and thus, special care is taken to work around this issue. When the lexer reads a `newline` symbol, it will go to a label `got_indents`. This place in the code will count the number of tabs in the line, and compare it to the number of tabs in the last line. If there is an equal number of tabs a `newline` token will be returned. If there is a difference in the number of tabs, a number of indent or dedent tokens equal to the difference will be returned. This is done to keep track of statements' place in scope - for example which statements are inside an `if` block or which class they belong to.

7.4 Minimisation of DFA's

A DFA built as described in section 7.2 does not necessarily result in the best DFA - it may be that several states can take the same input and end with the same accept rule. Merging equivalent states will result in a smaller DFA.

The DFA in figure 7.5 is such a case. After reading a *b* in state s1, no matter what input symbol is read only state s2 and s3 can be reached. As these two both are accepting states, they could be combined into one state, creating a smaller and more efficient DFA.

A notation for transitions on states and sets of states is introduced in definition 1.

> **Definition 1** Let s_1 and s_2 be states and $\mathbf{S_1}$ and $\mathbf{S_2}$ be sets of states, then transitions on the symbol a will be denoted:
>
> $$\begin{array}{lll} s_1 \overset{a}{\rightarrow} s_2 & \text{if} & \delta(s_1, a) = s_2 \\ s_1 \overset{a}{\rightarrow} \mathbf{S_2} & \text{if} & \exists s' \in \mathbf{S_2} \text{ where } s_1 \overset{a}{\rightarrow} s' \\ \mathbf{S_1} \overset{a}{\rightarrow} \mathbf{S_2} & \text{if} & \mathbf{S_2} = \{s' | \forall s'' \in \mathbf{S_1}, s'' \overset{a}{\rightarrow} s'\} \end{array}$$

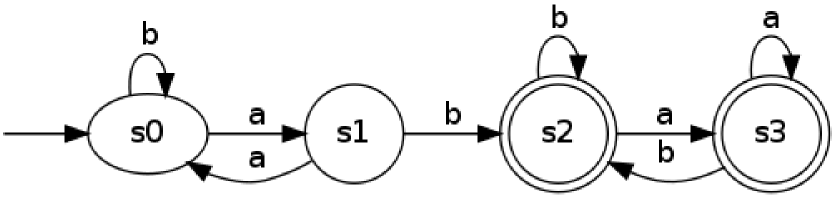


Figure 7.5: DFA which can be minimised.

7.4.1 Principle of Minimising a DFA

The method of determining which states should be merged is inspired by algorithm 3.39 given in Aho et al. [2006].

A set of states can be merged into one state if all their transitions lead to the same state or set of states.

The idea is therefore to create a family of sets of states, where the sets contain states that could potentially be merged, and then remove those that cannot. To determine which states that cannot be merged, a definition of equivalence of states is introduced in definition 2.

> **Definition 2** G is a family of sets of states, where $\forall \mathbf{S_i} \in G$ and $\forall a \in \Sigma$, then for some $\mathbf{S_i}$ there exist an $\mathbf{S_j}$ so that $\mathbf{S_i} \overset{a}{\to} \mathbf{S_j}$, $\{\mathbf{S_i}, \mathbf{S_j}\} \subseteq G$.
>
> Two states, s_k and s_h are $s_k \sim_G s_h$ if $\{s_k, s_h\} \subseteq \mathbf{S_i} \in G$.

Using this notation the states s_k and s_h can only be in the same set if $s_k \sim_G s_h$. When this applies to all states, we will have identified the states that can be merged.

In algorithm 1 this approach is listed as pseudo code. **Q** is the set of all states in the DFA.

Algorithm 1 Minimise(**Q**)

```
G := Create_family(Q)
while $\exists \mathrm{S_i} \in \mathbf{G}$ such that $\forall s' \in \mathbf{S_i}$ and $\forall s'' \in \mathbf{S_i}$, $s' \nsim_G s''$ do
  let $s \in \mathbf{S_i}$, and $\mathbf{S_k} = \emptyset$
  $\mathbf{S_i} := \mathbf{S_i} \setminus \{s\}, \mathbf{S_k} := \mathbf{S_k} \cup \{s\}$
  $\mathbf{G} := \mathbf{G} \setminus \mathbf{S_i}$
  $\mathbf{G} := \mathbf{G} \cup$ Split($\mathbf{S_k}$, $\mathbf{S_i}$)
end while
$\mathbf{Q} := \emptyset$
for $\forall \mathrm{S} \in \mathbf{G}$ do
  if $|\mathrm{S}| > 1$ then
    $\mathbf{Q} := \mathbf{Q} \cup$ Merge(Q, S)
  end if
  $\mathbf{Q} := \mathbf{Q} \cup S$
end for
return Q
```

7.4.2 Implementing Minimise(DFA)

Recall that all accepting states have an accept rule which denotes the output given if a string is accepted in that state. States with different accept rules cannot be merged, as that would result in a non-equivalent DFA. The initial sets will therefore consist of sets of states that either have the same accept rule or are non-accepting. In Create_family(**Q**) **Q** is the set of all states, $\mathbf{S_r}$ denotes a set of states which have the accept rule r, and **F** is the set of accept states, introduced in 7.1.

The next step is to split the sets so all states are equivalent. If two states s_i and s_j are $s_i \nsim_G s_j$, then s_j is removed to a new set of states, $\mathbf{S_k}$. The states of this set must also be equivalent. Split($\mathbf{S_1}$, $\mathbf{S_2}$) is the algorithm for splitting a set of states.

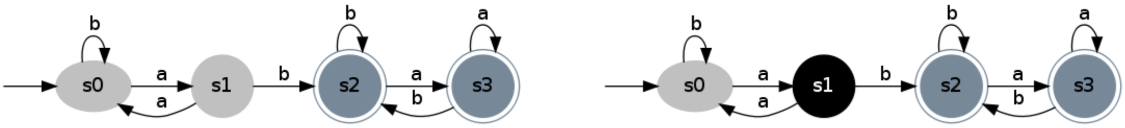
The last step is to merge the final sets. If the set **S**, contains more that one element, an arbitary state, s_1, is selected as the representative state. We must then move all transitions to states in **S** to this representative and delete the remaining states. But if the state from which the transition starts is already in **S**, both that state and the end state are changed to s_1. This is to prevent loss of transitions when states are later

Algorithm 2 Create_family($\mathbf{Q}$)

```
$\mathbf{G} := \{\mathbf{S_1}\}$
for each state $q_i \in \mathbf{Q}$ do
    r := nil
    if $q_i \in \mathbf{F}$ then
        r := accept rule
    end if
    if r = nil then
        $\mathbf{S_1} = \mathbf{S_1} \cup \{q_i\}$
    else if $\exists \mathbf{S_r} \in \mathbf{G}$ then
        $\mathbf{S_r} := \mathbf{S_r} \cup \{q_i\}$
    else
        create new set $\mathbf{S_r} := \{q_i\}$.
        Let $\mathbf{G} := \mathbf{G} \cup \mathbf{S_r}$.
    end if
end for
return $\mathbf{Q}$
```

Algorithm 3 Split($\mathbf{S_1}$, $\mathbf{S_2}$)

```
let $s_1 \in \mathbf{S_1}$
for all symbols $t \in \Sigma$ do
    let $\mathbf{S_j}$ be $\mathbf{S_j}$ in $s_1 \xrightarrow{t} \mathbf{S_j}$
    for all states $s_i \in \mathbf{S_2}$ do
        if $s_i \xrightarrow{t} \mathbf{S_j}$ then
            $\mathbf{S_2} := \mathbf{S_2} \setminus \{s_i\}$. $\mathbf{S_1} := \mathbf{S_1} \cup \{s_i\}$.
        end if
    end for
end for
return $\{\mathbf{S_1}, \mathbf{S_2}\}$
```



(a) DFA after creating the initial sets in algorithm 1

(b) DFA after $\mathbf{S_1}$ has been split.

Figure 7.6: Intermediate steps of minimising a DFA.

deleted. Algorithm 4 gives a pseudo code for this. **Q** is again the set of all states in the DFA.

Algorithm 4 Merge(Q, S)

```
let $s_1$ be some state of S
for all symbols $t \in \Sigma$ do
    for all states, $q_i$, in Q do
        if $q_i \in S$ and $q_i \xrightarrow{t} s \in$ S then
            $s_1 \xrightarrow{t} s_1$
        else if $q_i \xrightarrow{t} s \in$ S, where $s \neq s_1$ then
            $q_i \xrightarrow{t} s_1$
        end if
    end for
end for
return $\{s_i\}$
```

7.4.3 Example of minimising a DFA

Example 2 The DFA from figure 7.5 will be used to show how algorithm 1 can be implemented.

The first step is to create the family of sets of states. Using the accept rules, the two sets, $\mathbf{S_1}$ and $\mathbf{S_2}$ are created.
In figure 7.6a the states have been coloured to show this.

$$\mathbf{S_1} := \{s_0, s_1\}$$
$$\mathbf{S_2} := \{s_2, s_3\}$$

Then it is checked whether $\mathbf{S_1}$ must be split. The first symbol is a, but as both transitions lead to $\mathbf{S_1}$ no split is made.

$$s_0 \overset{a}{\rightarrow} \mathbf{S_1}$$
$$s_1 \overset{a}{\rightarrow} \mathbf{S_1}$$

For transitions on the next symbol, b, a split must be made, as s_1 leads to $\mathbf{S_2}$ but s_0 leads to $\mathbf{S_1}$ (see figure 7.6b).

$$s_0 \overset{b}{\rightarrow} \mathbf{S_1}$$
$$s_1 \overset{b}{\rightarrow} \mathbf{S_2}$$
$$S_1 := \{s_1\}$$
$$S_3 := \{s_0\}$$

$\mathbf{S_2}$ is the next set of states which is checked. Both s_2 and s_3 have a transition on a to $\mathbf{S_2}$ so no split is made.

$$s_2 \overset{a}{\rightarrow} \mathbf{S_2}$$
$$s_3 \overset{a}{\rightarrow} \mathbf{S_2}$$

Nor on b.

$$s_2 \overset{b}{\rightarrow} \mathbf{S_2}$$
$$s_3 \overset{b}{\rightarrow} \mathbf{S_2}$$

$\mathbf{S_3}$ contains only one state and cannot be split, so no sets can be split any further.

$\mathbf{S_1}$ and $\mathbf{S_3}$ contain only one state, so only $\mathbf{S_2}$ can be merged. s_2 is selected as the representative state.
All transitions to s_3 are moved to s_2, and all transitions from s_3 to $\mathbf{S_2}$ are changed to $s_2 \rightarrow s_2$.

$s_2 \overset{a}{\rightarrow} s_3$ changed to $s_2 \overset{a}{\rightarrow} s_2$
$s_3 \overset{a}{\rightarrow} s_3$ changed to $s_2 \overset{a}{\rightarrow} s_2$
$s_3 \overset{b}{\rightarrow} s_2$ changed to $s_2 \overset{b}{\rightarrow} s_2$

Figure 7.7 shows the resulting DFA.

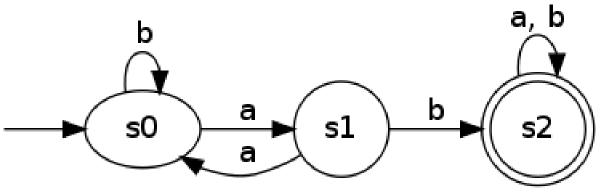


Figure 7.7: Minimised version of the DFA from figure 7.5.

7.5 Summary

The lexer generator uses regular expressions to recognise which tokens the lexer should return from which input. It builds a tree for each regular expression with the associated accept rule, after which it builds a DFA from a combined tree. As regular expressions and DFAs are equivalent, the states and transitions in the DFA is used to recognise which token should be returned.

When the lexer generator reads the file with regular expressions and accept rules, it builds the DFA, using switch statements and gotos. The generator reads a part of the input file into a buffer, from which it reads the regular expressions and accept rules. It represents states as labels, each having a switch statement, acting as the transition function according to the input symbol read while lexing. Each switch has a default goto, ensuring that a seperate switch for every possible input symbol is not necessary.

As the lexer generator does not necessarily produce an optimal DFA, an algorithm for minimising the DFA has been implemented. It sorts the states in to sets according to whether or not they are accepting states, and then sorts the accept states by accept rules. Then it is checked if all states in each set have the same transitions on every input symbol - if not, they are split. When there are no more splits to be made, the sets are shortened to being only a single state each, and the transitions to and from the sets are changed accordingly.

7.6 Lexical Analysis Benchmark

This section describes a benchmark test performed on our lexical analyser generator, Lexter, versus GNU flex, a fast lexical analyser generator written in C. Both lexers were given an approximately 1GB PL/0 code file to tokenize. This amounts to approximately 90.000.000 (90 million) lines of code, of which only 38 lines are unique. They were given the same file to analyse. The lexers increment 4 different counters for each token they read, depending on if the token is a keyword, value, operator, or id. The time required to tokenize the code file is measured in seconds.

As figure 7.8 shows, Lexter performs 20 % better then flex under this test. The figure reflects time spent in user space, as both lexers used more or less the same amount of time in system space. Both lexers run in O(n). Furthermore, Lexter also has a smaller binary file size compared to flex.

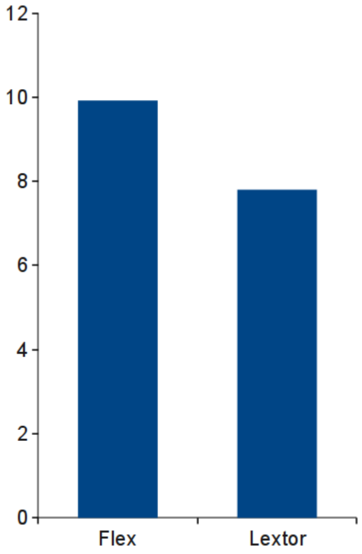


Figure 7.8: Lexter vs flex tokenize challenge.

CHAPTER 8

Parsing

An introduction to LR parsing, with focus on the implementation of our LALR(1) parser generators: g2c.

8.1 Context-Free Grammars

A context-free grammar (CFG) is used to specify the precise syntactic structure of a programming language by stating a number of substitution rules for each language construct, which consist of two kinds of symbols: terminals and nonterminals. Terminals are the basic symbols from which strings can be formed. Nonterminals denote sets of strings and are used to enforce a hierarchical structure on the language generated by the grammar. A CFG has four components - making it a 4-tuple (V, Σ, R, S) [Aho et al., 2006] where

- V is a finite set of nonterminals. These are also known as syntactic variables.
- Σ is a set of terminal symbols.
- R is a set of substitution rules or, rather, productions. Each rule is a nonterminal, followed by a string of mixed nonterminals and terminals.
- $S \in V$ is a start symbol, which is one of the nonterminals.

Productions

A production is a rule of the grammar. As stated, each production starts with a nonterminal, named the head or left hand side, an arrow, and a string of mixed terminals and nonterminals. This is called the body or right hand side of the production. The production specifies a substitution rule of how a construct can be written. The head nonterminal represents a construct and the body represents the written form of the construct.

$$\begin{array}{rcl} \langle expr \rangle & \rightarrow & \langle expr \rangle \; \textbf{'+'} \; \langle term \rangle | \langle term \rangle \\ \langle term \rangle & \rightarrow & \langle term \rangle \; \textbf{'*'} \; \langle factor \rangle | \langle factor \rangle \\ \langle factor \rangle & \rightarrow & \textbf{'('} \; \langle expr \rangle \; \textbf{')'} \mid \textbf{'id'} \end{array}$$

Derivations

A derivation of a grammar is a sequence of substitution steps required in order to obtain a string. The process of derivation begins with the start symbol, which in turn is substituted with the body of the next possible substitution rule until no further derivation steps can be performed. It is therefore possible to follow the steps required to generate a given string in the language of a grammar.

$$u \overset{*}{\Rightarrow} v$$

Derivations can be performed in two ways. The first method is called the leftmost derivation, which means that the leftmost nonterminal in the body is derived before the next nonterminal. At each step the current nonterminal under consideration is replaced by the next substitution rule. Once a number of derivation steps have been performed the appropriate terminals will have replaced the leftmost nonterminal and then the next nonterminal follows the same pattern until a string of terminals is yielded.

A rightmost derivation proceeds by expanding the nonterminal farthest to the right of the string, until there are no more production rules that can be derived.

8.2 LR Parsing

An introduction to the LR parsing algorithm.

Parsing is the process of reading a sequence of tokens and constructing a parse tree by deducing the sequence of productions of the grammar that were used to generate the sequence of input tokens. An LR parser is a bottom-up left to right parser that produces the rightmost derivation. This means that it reads the input left to right, and builds the parse tree bottom-up, always producing the rightmost derivation. Compared to LL parsers, LR parsers can recognize a larger subset of the context free languages. LR parsers can be generated from context free grammars (CFGs) with left recursion [Aho et al., 2006].

An LR parser can be implemented as a deterministic Push-Down Automaton (PDA), which has a set of states, a stack, a stack alphabet and an input alphabet. An LR parser reads terminals from the lexer and depending on state and input terminals, it either pushes states and terminals on to the stack, reduces symboles on the stack or returns the top of the stack as the result.

An LR parser can perform three types of actions:

Shift s Push the current terminal and the state s onto the stack and move to the next state.

Reduce $\langle A\rangle \rightarrow \langle B\rangle$ Pop $|\langle B\rangle|$ symbols and states of the stack and use them to produce $\langle A\rangle s$. Then consult the GOTO table entry for the topmost state of the stack and push the state for $\langle A\rangle$ on to the stack along with $\langle A\rangle$.

Accept Return the topmost symbol from the stack as the result.

An LR parser consults the ACTION table to determine which action to perform given the topmost/current state and current input terminal. Table 8.1 is an example of the ACTION and GOTO tables of an LR parser for grammar 1. ″s, a″ where a is

$$\begin{aligned}
\langle Expr\rangle &\rightarrow \langle Expr\rangle\ \textbf{'+'}\ \langle Term\rangle && (8.1)\\
&\mid\ \langle Term\rangle && (8.2)\\
\langle Term\rangle &\rightarrow \langle Term\rangle\ \textbf{'*'}\ \langle Factor\rangle && (8.3)\\
&\mid\ \langle Factor\rangle && (8.4)\\
\langle Factor\rangle &\rightarrow \textbf{'('}\ \langle Expr\rangle\ \textbf{')'} && (8.5)\\
&\mid\ \textbf{number} && (8.6)
\end{aligned}$$

Grammar 1: Grammar for basic arithmetics.

	ACTION						GOTO		
State	**EOF**	′+′	′*′	**′(′**	**′)′**	**number**	$\langle Expr\rangle$	$\langle Term\rangle$	$\langle Factor\rangle$
0				s, 4		s, 5	1	2	3
1	Accept	s, 6							
2	r, 8.2	r, 8.2	s, 7		r, 8.2				
3	r, 8.4	r, 8.4	r, 8.4		r, 8.4				
4				s, 4		s, 5	8	2	3
5	r, 8.6	r, 8.6	r, 8.6		r, 8.6				
6				s, 4		s, 5		9	3
7				s, 4		s, 5			10
8		s, 6			s, 11				
9	r, 8.1	r, 8.1	s, 7		r, 8.1				
10	r, 8.3	r, 8.3	r, 8.3		r, 8.3				
11	r, 8.5	r, 8.5	r, 8.5		r, 8.5				

Table 8.1: LR(1) parser table for grammar 1.

a digit means ″Shift a″, in the above context and ″r, x″ means ″Reduce″ where x is a reference to the production, from the grammar, which should be reduced.

An LR parser starts in its start state (that is state 0), reads a terminal, consults the ACTION table and executes actions until the accept action is reached at which point it returns. Notice that the ACTION table shows which action to perform and the GOTO tables shows which state to enter, having reduced to a non-terminal.

There are different variants of LR parsers. The only difference between these parsers is how their ACTION and GOTO tables are generated. On figure **??** the parsers and the subsets of the unambiguous context free languages they can recognise is illustrated. This figure shows that an LL(0) parser can only recognise a subset of the context free languages an LR(0) parser can recognise. The ″(0)″, ″(1)″ and ″(k)″ denotes the number of lookahead symbols the parser uses. When choosing which parser to use it is important to understand that the recognition power comes at the cost of larger parser tables.

We have chosen to write an LALR(1) parser generator for groo, because LALR(1) offers a good ratio between power and size. And by building our own parser generator we expect to provide a fairly good performance too. Doing a hand written parser

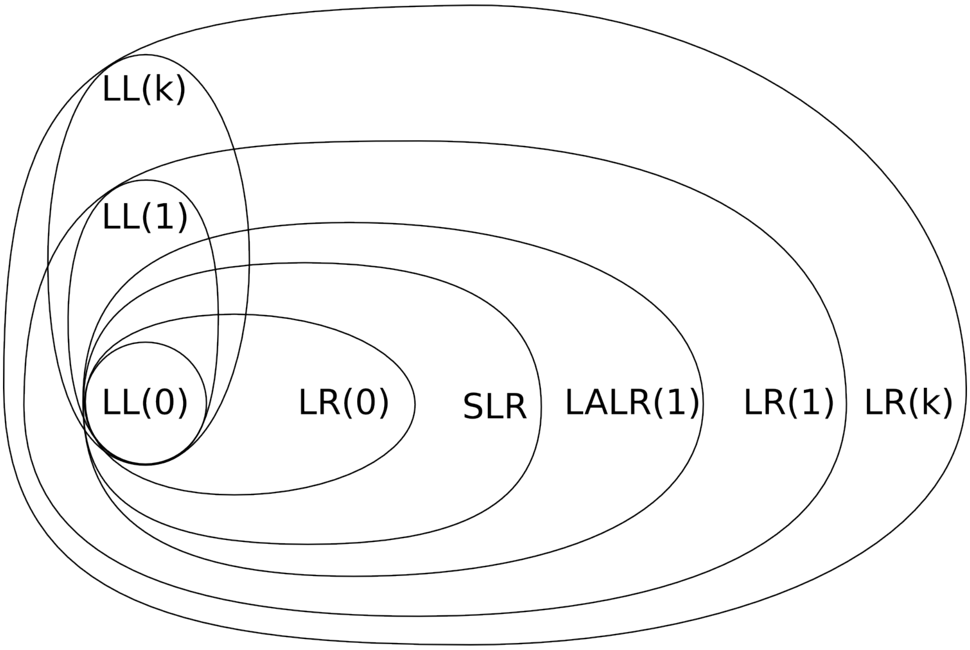


Figure 8.1: Classes of context free grammars.

would be possible, if we wanted to settle with an LL parser, however, this would not be very flexible and it would make it difficult to add new language constructs later in the development.

8.3 LALR(1) Table Generation

The set of states of an LR parser is a set of items. For a parser without lookaheads, an item is a tuple of a production and a position within this production. We shall call this an LR(0) item. For parsers with lookaheads, an item is a tuple of a production, a position within the production and a lookahead symbol; we shall call this an LR(1) item.

For convenience we denote an LR(0) item as a production with a dot, denoting the position within the production. For example: $\{\langle Expr\rangle \rightarrow \langle Expr\rangle \cdot$ **'+'** $\langle Term\rangle\}$ is an LR(0) item. LR(1) items are denoted as follows: $\{\langle Expr\rangle \rightarrow \langle Expr\rangle \cdot$ **'+'** $\langle Term\rangle,$ **'*'** $\}$ where **'*'** is the lookahead symbol.

> **Definition 3** The core of an LR(1) item $\{\langle A\rangle \rightarrow \alpha \cdot \beta, X\}$ is the LR(0) item $\{\langle A\rangle \rightarrow \alpha \cdot \beta\}$.

A state is a set of items, where the position within the items denotes the parts of a possible production read so far. To compute an LALR(1) parser table, we will need to compute the LALR(1) states, and the sets of LR(1) items that denote these states. This can be done by computing all LR(1) states, and merging all states with matching cores (see definition 3), which is an expensive operation due to the number of LR(1) states. Alternately, the LALR(1) states can be found by computing the LR(0) states and computing appropriate lookaheads for these.

Definition 4 An LR(0) or LR(1) item $\{\langle A \rangle \rightarrow \alpha \cdot \beta, a\}$ is a kernel item if α is a non-empty sequence of symbols or $\langle A \rangle$ is the start symbol in the grammar.

8.3.1 Computation of LR(0) States

To compute all the LR(0) states for a context free grammar we introduce three functions: $CLOSURE_{LR(0)}$, $GOTO_{LR(0)}$ and $items_{LR(0)}$. The $CLOSURE_{LR(0)}$ function can compute the set of items that denote a state given the kernel items for this state, as seen in algorithm 5, which is inspired by Aho et al. [2006, fig. 4.32].

Algorithm 5 Compute $CLOSURE_{LR(0)}(I, G)$ for LR(0) items.

```
Input: A CFG $G$ and a set of LR(0) items $I$
  $retval = I$
  $new = stack(I)$ {This initialize $new$ to be a stack}
  while $|new| \neq 0$ do
    $n = new.pop()$
    for all productions $\{\langle B \rangle \rightarrow \gamma\} \in G$ where $n = \{\langle A \rangle \rightarrow \alpha \cdot \langle B \rangle \beta\}$ do
      {$\alpha$, $\beta$ and $\gamma$ are sequences of zero or more symbols.}
      if $\{\langle B \rangle \rightarrow \cdot\gamma\} \notin retval$ then
        $retval = retval \cup \{\langle B \rangle \rightarrow \cdot\gamma\}$
        $new.push(\{\langle B \rangle \rightarrow \cdot\gamma\})$
      end if
    end for
  end while
  return $retval$
```

When we have a set of items I that constitue a state, the $GOTO$ function [Aho et al., 2006, section 4.6.2] can compute the set of items $GOTO$ that denote a state to which we must go when X is read or reduced in state I.

Algorithm 6 Compute $GOTO_{LR(0)}(I, X, G)$ for LR(0) items.

```
Input: $I$ a set of LR(0) items, a symbol $X \in \Sigma$ and a CFG $G = (V, \Sigma, R, S)$
  $retval = \{\}$
  for all $\{\langle A \rangle \rightarrow \alpha \cdot X\beta\} \in I$ do
    {$\alpha$ and $\beta$ are sequences of zero or more symbols.}
    $retval = retval \cup \{\langle A \rangle \rightarrow \alpha X \cdot \beta\}$
  end for
  return $CLOSURE(retval, G)$
```

If we have a context free grammar G and wish to find all the LR(0) states, we just need to find the set of items that denotes the start state. Then it is just a matter of applying the $GOTO_{LR(0)}$ function, until all states have been found. To easily find the set of items that denotes the start state, we augment the grammar with a new start symbol, that is only used in one production.

> **Definition 5** If $\langle S' \rangle \in G.V$ is the start symbol of the grammar G, then $G' = (V', \Sigma, R', S)$ is the augmented grammar, where $V' = G.V \cup \{\langle S' \rangle\}$, $R' = R \cup \{\langle S' \rangle \to \langle S \rangle\}$ and $G.S = \langle S' \rangle$, where $\langle S' \rangle \notin G.V \cup G.\Sigma$.

When we have an augmented grammar G' we can be certain that the set of items, denoting the start state is $CLOSURE(\{\langle S' \rangle \to \cdot\langle S \rangle\}, G')$ because $\langle S' \rangle \to \langle S \rangle$ is the only production for $\langle S' \rangle$, which is the start symbol. The items function, algorithm 7 inspired by Aho et al. [2006, Fig. 4.33] but optimized with a stack, takes an augmented grammar and computes the sets of items denoting the LR(0) states.

Algorithm 7 Compute $items_{LR(0)}(G')$ for LR(0) items.

```
Input: CFG $G'$ with augmented with start variable $\langle S' \rangle$
  $states = \{CLOSURE(\{\langle S' \rangle \to \cdot\langle S \rangle\}, G')\}$
  $new = stack(states)$ {This initialize $new$ to be a stack}
  while $|new| \neq 0$ do
    $I = new.pop()$
    for all grammar symbols $X \in V \cup \Sigma$ where $G' = (V, \Sigma, R, S)$ do
      $g = GOTO(I, X, G')$
      if $|g| \neq 0 \wedge g \notin states$ then
        $states = states \cup \{g\}$
        $new.push(g)$
      end if
    end for
  end while
  return $states$
```

8.3.2 Computation of LR(1) States

To compute LR(1) states for a CFG we will introduce the 3 functions: $CLOSURE_{LR(1)}$, $GOTO_{LR(1)}$ and $items_{LR(1)}$ for LR(1) items. Again the $CLOSURE_{LR(1)}$ function, algorithm 8 inspired by Aho et al. [2006] but optimized for performance with a stack, can compute the set of items that denotes a state given the kernel items for this state.

Algorithm 8 Compute $CLOSURE_{LR(1)}(I, G)$ for LR(1) items.

```
Input: A CFG $G$ and a set of LR(1) items $I$
  $retval = I$
  $new = stack(I)$ {This initialize $new$ to be a stack}
  while $|new| \neq 0$ do
    $n = new.pop()$
    for all productions $\{\langle B\rangle \to \gamma\} \in G$ where $n = \{\langle A\rangle \to \alpha \cdot \langle B\rangle\beta, x\}$ do
      {$\alpha$, $\beta$ and $\gamma$ are sequences of zero or more symbols.}
      for all terminal $y \in FIRST(\beta x)$ do
        {$FIRST(\langle X\rangle)$ is the set of terminals $a \in FIRST(\langle X\rangle)$ such that $\langle X\rangle \overset{*}{\Rightarrow} a\delta$
        where $\delta$ is a sequance of zero or more terminals.}
        if $\{\langle B\rangle \to \cdot\gamma, y\} \notin retval$ then
          $retval = retval \cup \{\langle B\rangle \to \cdot\gamma, y\}$
          $new.push(\{\langle B\rangle \to \cdot\gamma, y\})$
        end if
      end for
    end for
  end while
  return $retval$
```

The $GOTO_{LR(1)}$ function, algorithm 9 from Aho et al. [2006, section 4.40], can given a set of items, a symbol and a grammar, compute the set of items that denotes the next state the parser should enter having read or reduced the given terminal.

Algorithm 9 Compute $GOTO_{LR(1)}(I, X, G)$ for LR(1) items.

```
Input: $I$ a set of LR(1) items, a symbol $X \in \Sigma$ and a CFG $G = (V, \Sigma, R, S)$
  $retval = \{\}$
  for all $\{\langle A\rangle \to \alpha \cdot X\beta, a\} \in I$ do
    {$\alpha$ and $\beta$ are sequences of zero or more symbols.}
    $retval = retval \cup \{\langle A\rangle \to \alpha X \cdot \beta, a\}$
  end for
  return $CLOSURE_{LR(1)}(retval, G)$
```

The items function, algorithm 10, can given an augmented CFG compute all sets of items. This is done using the $GOTO_{LR(1)}$ function from any set of items found, until no new sets of items can be found. Thus all states will have been found.

Algorithm 10 Compute $items_{LR(1)}(G')$ for LR(1) items.

```
Input: CFG $G'$ with augmented start variable $\langle S' \rangle$ and '$' denotes end of input.
  $states = \{CLOSURE(\{\langle S' \rangle \to \cdot \langle S \rangle, \text{'\$'}\}, G')\}$
  $new = stack(states)$ {This initialize $new$ to be a stack}
  while $|new| \neq 0$ do
    $I = new.pop()$
    for all grammar symbols $X \in V \cup \Sigma$ where $G' = (V, \Sigma, R, S)$ do
      $g = GOTO(I, X, G')$
      if $|g| \neq 0 \wedge g \notin states$ then
        $states = states \cup \{g\}$
        $new.push(g)$
      end if
    end for
  end while
  return $states$
```

8.3.3 Computation of LALR(1) States

LALR(1) states for a grammar G is effectively the LR(1) states for the grammar G where all states with the same core (by defintion 3) have been merged. Thus it is possible, however, very inefficient to find the LALR(1) states by computing the LR(1) states and merging them as required. But as previously explained it is also possible to compute the LR(0) states and find lookaheads for these. This is done in two steps, first we compute spontaneous lookaheads and lookahead propagation, then we propagate the spontaneous lookaheads to states using the information about lookahead propagation.

Algorithm 11 Finding spontaneous and propagated lookaheads

```
Input: CFG $G$ and the set of kernel items $K$ for the set of items that denotes an LR(0)
  state of $G$
  $I = CLOSURE_{LR(0)}(K, G)$
  for all items $\{\langle A \rangle \to \alpha \cdot \beta\} \in K$ do
    $J = CLOSURE_{LR(1)}(\{\{\langle A \rangle \to \alpha \cdot \beta, \#\}\}, G)$ where $\# \notin \Sigma \cup V$ and $G = (V, \Sigma, R, S)$

    if $\{\langle B \rangle \to \gamma \cdot X\delta, a\} \in J \wedge a \neq \#$ then
      $a$ is a spontaneously generated lookahead for
      $\{\langle B \rangle \to \gamma X \cdot \delta\} \in GOTO_{LR(0)}(I, X, G)$
    end if
    if $\{\langle B \rangle \to \gamma \cdot X\delta, \#\} \in J$ then
      Lookaheads propagate from $\langle A \rangle \to \alpha \cdot \beta \in I$ to
      $\langle B \rangle \to \gamma X \cdot \delta \in GOTO_{LR(0)}(I, X, G)$
    end if
  end for
```

Algorithm 11, Aho et al. [2006, Algorithm 4.62], can be used to find spontaneous lookaheads and lookahead propagation in a grammar. The algorithm must be executed for each set of LR(0) kernel items that denote a state. In a practical implementation the result can be stored in a table.

Table 8.2 shows the result of running algorithm 11 for all sets of LR(0) kernels that denote a state of the augmented grammar of grammar 1. In table 8.2 there is a row for each kernel, and each kernel has been assigned an id[1]. The column "Propagation ids" contains a set of ids that the kernel propagates lookaheads to.

Id	State	Item	Sp. lookaheads	Propagation ids
0	I_0:	$\langle S'\rangle \rightarrow \cdot\langle Expr\rangle$	$\{\,\textbf{'\$'}\,\}$	$\{1,2,3,4,5,6,7\}$
1	I_1:	$\langle S'\rangle \rightarrow \langle Expr\rangle\cdot$	$\{\}$	$\{\}$
2		$\langle Expr\rangle \rightarrow \langle Expr\rangle \cdot \textbf{'+'}$	$\{\,\textbf{'+'}\,\}$	$\{8\}$
3	I_2:	$\langle Expr\rangle \rightarrow \langle Term\rangle\cdot$	$\{\,\textbf{'+'}\,,\,\textbf{')'}\,\}$	$\{\}$
4		$\langle Term\rangle \rightarrow \langle Term\rangle \cdot \textbf{'*'}\,\langle Factor\rangle$	$\{\,\textbf{'+'}\,,\,\textbf{'*'}\,,\,\textbf{')'}\,\}$	$\{9\}$
5	I_3:	$\langle Term\rangle \rightarrow \langle Factor\rangle\cdot$	$\{\,\textbf{'+'}\,,\,\textbf{'*'}\,,\,\textbf{')'}\,\}$	$\{\}$
6	I_4:	$\langle Factor\rangle \rightarrow \textbf{'('}\,\cdot\langle Expr\rangle\,\textbf{')'}$	$\{\,\textbf{'+'}\,,\,\textbf{'*'}\,,\,\textbf{')'}\,\}$	$\{10\}$
7	I_5:	$\langle Factor\rangle \rightarrow \textbf{number}\,\cdot$	$\{\,\textbf{'+'}\,,\,\textbf{'*'}\,,\,\textbf{')'}\,\}$	$\{\}$
8	I_6:	$\langle Expr\rangle \rightarrow \langle Expr\rangle\,\textbf{'+'}\,\cdot\langle Term\rangle$	$\{\}$	$\{5,6,7,12,13\}$
9	I_7:	$\langle Term\rangle \rightarrow \langle Term\rangle\,\textbf{'*'}\,\cdot\langle Factor\rangle$	$\{\}$	$\{6,7,14\}$
10	I_8:	$\langle Factor\rangle \rightarrow \textbf{'('}\,\langle Expr\rangle\cdot\,\textbf{')'}$	$\{\}$	$\{15\}$
11		$\langle Expr\rangle \rightarrow \langle Expr\rangle\cdot\,\textbf{'+'}\,\langle Term\rangle$	$\{\,\textbf{'+'}\,,\,\textbf{')'}\,\}$	$\{8\}$
12	I_9:	$\langle Expr\rangle \rightarrow \langle Expr\rangle\,\textbf{'+'}\,\langle Term\rangle\cdot$	$\{\}$	$\{\}$
13		$\langle Term\rangle \rightarrow \langle Term\rangle\cdot\,\textbf{'*'}\,\langle Factor\rangle$	$\{\,\textbf{'*'}\,\}$	$\{9\}$
14	I_{10}:	$\langle Term\rangle \rightarrow \langle Term\rangle\,\textbf{'*'}\,\langle Factor\rangle\cdot$	$\{\}$	$\{\}$
15	I_{11}:	$\langle Factor\rangle \rightarrow \textbf{'('}\,\langle Expr\rangle\,\textbf{')'}\,\cdot$	$\{\}$	$\{\}$

Table 8.2: Result of running algorithm 11 for the augmented grammar of grammar 1

Once a table like table 8.2 has been computed, the lookaheads should be propagated. This is done by adding the set of lookaheads for one item to the sets of lookaheads for the items i propagates to. This is done for all items and repeated untill no more propagation occurs. The result of doing this for table 8.2 can be seen in table 8.3.

Following table 8.3 the LALR(1) kernel items of state I_1 are $\{\langle S'\rangle \rightarrow \langle Expr\rangle\cdot,\ \textbf{'\$'}\,\}$, $\{\langle Expr\rangle \rightarrow \langle Expr\rangle\cdot\ \textbf{'+'}\,,\ \textbf{'+'}\,\}$ and $\{\langle Expr\rangle \rightarrow \langle Expr\rangle\cdot\ \textbf{'+'}\,,\ \textbf{'\$'}\,\}$. The entire set of items for state I_1 can be found using $CLOSURE_{LR(1)}(I, G)$ from algorithm 8.

[1] The ids are unique and have only been introduced for the notational convenience.

State	**Item**	**Lookaheads**
I_0:	$\langle S' \rangle \rightarrow \cdot\langle Expr \rangle$	$\{ \mathbf{'\$'} \}$
I_1:	$\langle S' \rangle \rightarrow \langle Expr \rangle\cdot$	$\{ \mathbf{'\$'} \}$
	$\langle Expr \rangle \rightarrow \langle Expr \rangle \cdot \mathbf{'+'}$	$\{ \mathbf{'+'} , \mathbf{'\$'} \}$
I_2:	$\langle Expr \rangle \rightarrow \langle Term \rangle\cdot$	$\{ \mathbf{'+'} , \mathbf{')'} , \mathbf{'\$'} \}$
	$\langle Term \rangle \rightarrow \langle Term \rangle \cdot \mathbf{'*'} \langle Factor \rangle$	$\{ \mathbf{'+'} , \mathbf{'*'} , \mathbf{')'} , \mathbf{'\$'} \}$
I_3:	$\langle Term \rangle \rightarrow \langle Factor \rangle\cdot$	$\{ \mathbf{'+'} , \mathbf{'*'} , \mathbf{')'} , \mathbf{'\$'} \}$
I_4:	$\langle Factor \rangle \rightarrow \mathbf{'('} \cdot \langle Expr \rangle \mathbf{')'}$	$\{ \mathbf{'+'} , \mathbf{'*'} , \mathbf{')'} , \mathbf{'\$'} \}$
I_5:	$\langle Factor \rangle \rightarrow \textbf{number} \cdot$	$\{ \mathbf{'+'} , \mathbf{'*'} , \mathbf{')'} , \mathbf{'\$'} \}$
I_6:	$\langle Expr \rangle \rightarrow \langle Expr \rangle \mathbf{'+'} \cdot \langle Term \rangle$	$\{ \mathbf{'+'} , \mathbf{')'} , \mathbf{'\$'} \}$
I_7:	$\langle Term \rangle \rightarrow \langle Term \rangle \mathbf{'*'} \cdot \langle Factor \rangle$	$\{ \mathbf{'+'} , \mathbf{'*'} , \mathbf{')'} , \mathbf{'\$'} \}$
I_8:	$\langle Factor \rangle \rightarrow \mathbf{'('} \langle Expr \rangle \cdot \mathbf{')'}$	$\{ \mathbf{'+'} , \mathbf{'*'} , \mathbf{')'} , \mathbf{'\$'} \}$
	$\langle Expr \rangle \rightarrow \langle Expr \rangle \cdot \mathbf{'+'} \langle Term \rangle$	$\{ \mathbf{'+'} , \mathbf{')'} \}$
I_9:	$\langle Expr \rangle \rightarrow \langle Expr \rangle \mathbf{'+'} \langle Term \rangle\cdot$	$\{ \mathbf{'+'} , \mathbf{')'} , \mathbf{'\$'} \}$
	$\langle Term \rangle \rightarrow \langle Term \rangle \cdot \mathbf{'*'} \langle Factor \rangle$	$\{ \mathbf{'*'} , \mathbf{'+'} , \mathbf{')'} , \mathbf{'\$'} \}$
I_{10}:	$\langle Term \rangle \rightarrow \langle Term \rangle \mathbf{'*'} \langle Factor \rangle\cdot$	$\{ \mathbf{'+'} , \mathbf{'*'} , \mathbf{')'} , \mathbf{'\$'} \}$
I_{11}:	$\langle Factor \rangle \rightarrow \mathbf{'('} \langle Expr \rangle \mathbf{')'} \cdot$	$\{ \mathbf{'+'} , \mathbf{'*'} , \mathbf{')'} , \mathbf{'\$'} \}$

Table 8.3: Result of propagating the lookaheads of table 8.2.

8.3.4 Generating Parsing Tabels

To generate the LALR(1) parsing tabels ACTION and GOTO for a grammar G the sets of items $\{I_0, I_1, I_2 \ldots\}$ that denotes the states of the augmented grammar G' will be needed. They can be computed as shown in section 8.3.3. The parsing tables can be computed by using algorithm 12, from Aho et al. [2006, algorithm 4.56].

Algorithm 12 Building parser table

```
Input: The sets of items that denotes the states $\{I_0, I_1 \ldots\}$ of an augmented CFG $G'$
  for all each set of items $I_i$ that denotes an LALR(1) state of $G'$ do
    for all $\{\langle A\rangle \to \alpha \cdot$ 'a' $\beta,$ 'b' $\} \in I_i$ do
      $I_j = GOTO_{LR(1)}(I_i,$ 'a' $, G')$
      $ACTION[i,$ 'a' $]$ = "shift $j$"
    end for
    for all $\{\langle A\rangle \to \alpha\cdot,$ 'b' $\} \in I_i$ do
      if $\langle A\rangle \neq \langle S'\rangle$ then
        $ACTION[i,$ 'a' $]$ = "reduce $\langle A\rangle \to \alpha$"
      else
        $ACTION[i,$ '\$' $]$ = "accept" {In this case 'a' = '\$' }
      end if
    end for
    for all $\langle A\rangle \in V$ where $G' = (V, \Sigma, R, S)$ do
      $I_j = GOTO_{LR(1)}(I_i, \langle A\rangle, G')$
      $GOTO[i, \langle A\rangle] = j$
    end for
  end for
```

Note that if there's is a conflict in during the generation of the parsing tables, using algorithm 12, the grammar is not in the LALR(1) class of context free grammars. Imagine that given a state and an input terminal the $ACTION$ table doesn't know whether to shift or reduce. To resolve such a situation the grammar could be transformed. Alternatively, conflicts can also be addressed with a conflict resolution strategy, as presented in section 8.4.

8.4 Resolution of Conflicts

As mentioned in section 8.3.4 a conflict resolution strategy can be used to resolve action conflicts and generate LALR(1) parsing tables for a grammar that is not in the LALR(1) class of context free grammars.

There exists three different kinds of conflicts, shift-shift, shift-reduce and reduce-reduce. A shift-shift conflict occurs when algorithm 12 cannot determine which state to enter having shifted the input terminal. A shift-reduce conflict occurs if algorithm 12 cannot determine whether to shift or reduce given input and state. A reduce-reduce conflict occurs if algorithm 12 cannot determine which reduction to perform.

A conflict resolution strategy specifies how to resolve these conflicts. In many parser generators, such as GNU bison, shift-reduce conflicts are implicitly resolved in favor of shift, whereas other conflicts requires annotations to the grammar, in form of precedence and associativity.

In our parser generator g2c, conflicts are always implicitly resolved in favor of the first mentioned grammar rule and in a shift-reduce conflict between a shift and a reduce using the same rule, the reduction is prefered. This strategy is quite easily implemented and keeps that grammar clear of ugly annotations.

This strategy also means that conflicts occure quietly, and the developer risks forgetting about them. However, g2c prints all conflicts to a logfile, and alerts the developer if there were conflicts. In most cases checking that the conflicts in the logfile are resolved as desired is sufficient, if not the input grammar rules can be rearranged as needed.

This conflict resolution strategy is probably not desirable for very complex grammars, and may not offer as much power as conflict resolutions strategies that uses annotations. Nevertheless, it's easy to implement, and easily resolves the conflicts we have allowed in our grammar.

8.5 Error Recovery

When the parser can neither shift, reduce or accept it enters an error mode. When in error mode, the parser interpretes the unexpected terminal as an ErrorTerminal. The parser will then remove symbols from the stack, until it enters a state where the error symbol can be shifted. If it is unable to find a state where the error symbol can be shifted, it will terminate. The parser must then delete enough tokens from the input, so it can successfully continue to parse without generating further errors.

This method of error recovery is known as "Panic Mode", and is easy to implement because it is often easy to recognise the end of a statement. If the parser can end a statement legally with an error terminal present, it can move on to continue parsing the rest of the application. This is also the main reason it was chosen for our parser.

It is also possible to implement what is known as "Phrase-Level recovery". This approach differs in that the parser will attempt to identify the intent of the erroneous code, and then, if a subroutine is available for correcting the error, correct it. Such subroutines could insert, remove or alter the input stack to allow it to continue parsing. There is a potential risk here however for entering an infinite loop, so such actions must be considered carefully. [Aho et al., 2006]

8.6 Efficient Push-Down Automaton Implementation

This sections discusses how to efficiently implement a deterministic Push-Down Automaton (PDA) in C++ with GCC extensions.

Many parser generators such as Bison implement the PDA of the parser using a table in memory. That is, a multidimensional array from which an action can be found in $O(1)$ using a state and terminal. This may seem like a good solution, however, using GCC extensions it is possible to implement a PDA directly in C++, similar to how we implemented the DFA in the lexer. This is interesting as it may offer greater performance.

The basic idea is to make labels for all actions and states, then push terminals and label pointers on to a stack, while jumping between the different actions using goto statements. In the following code samples we are using two stacks, one for symbols and one for states, however, as they are being pushed and popped simultaneously combining them should be easy. Also notice that instead of using heap allocations for these stacks we could use stack allocations if C++ would allow this.

The parse function takes a lexer as argument and start by initializing it as two stacks, a variable for topmost non-terminal, a pointer to the current terminal and some temporary variables for action code execution. Then it jumps to the start state, `START` which just an alias for the label `STATE0`.

```
Node* parse(Lexer* lexer){
    stack<Node*> symbols = stack<Node*>();
    stack<void*> states = stack<void*>();
    Symbol top_non_terminal; //Topmost non−terminal
    Terminal* current_terminal = lexer−>nextToken();
    Node* result, *arg0, *arg2, *arg1; // Used in actioncode
    goto START;
```

Each state in the PDA is implemented similar to this. Please note that the code below is an example of the structure, not an example of an actual state. If the program jumps to the `Shift5` label, a shift action will be executed. It is quite obvious that the current terminal is pushed to the symbols stack and that a new terminal is read. It is however, less obvious what the `states.push(&&Goto5);` statement does. This statement uses a GCC extention to get a pointer to the label `Goto5` and pushes it to a stack. The only reason our LR parser needs the current state to be pushed on to the stack is that it may need to perform lookups using the GOTO table entry for this state after a future reduction. So pushing a pointer to the label where the GOTO table entry for this state is implemented as a switch solves this issue quite nicely. Also notice that the `State5` label is placed before the label pointer is pushed to the `states` stack, this is because the action for each case in the GOTO table entry would otherwise require the label pointer to be pushed here.

```
Shift5:
    symbols.push(current_terminal);
    current_terminal = lexer−>nextToken();
State5:
    states.push(&&Goto5);
    switch(current_terminal−>token()){
        case TOKEN_EOF: goto Reduce1;
        case TOKEN_minus: goto Shift8;
        case TOKEN_plus: goto Shift10;
        default: goto ERROR;
    }
Goto5:
    switch(top_non_terminal){
        case SYMBOL_E: goto State18;
        case SYMBOL_F: goto State4;
        case SYMBOL_error:
            states. pop();
            symbols.pop();
            goto *states.top();
        default: goto FATAL_ERROR;
    }
```

For all productions a reduction action is implemented. Below is an example of how these are implemented by our parser generator. The parser assigns symbols to a variable such as `arg0` which may be used in the action code, or deletes them. It then pops symbols and states off the stacks. The action code sets the `result` variable to the result. This result is then pushed on to the stack, and the topmost non-terminal variable `top_non_terminal` is set, before the topmost GOTO entry label pointer is dereferenced and jumped to. Which handles GOTO table lookup and jumps back into a state.

```
Reduce7: //E −> E [plus] E
    arg2 = symbols.top();
    symbols.pop();
    states. pop();
    delete symbols.top();
    symbols.pop();
    states. pop();
    arg0 = symbols.top();
    symbols.pop();
    states. pop();
    result  = NULL;
    {
        result  = new AddNode(arg0, arg2);
    }
    symbols.push(result);
    top_non_terminal = SYMBOL_E;
    goto *states.top();
```

The parser will jump to the `ERROR` label if it encounters an unexpected terminal. At the `ERROR` label the current terminal is wrapped in an `ErrorTerminal` and the topmost non-terminal is set to error. Then the program jumps to the GOTO table entry for the topmost state on the `states` stack.

```
ERROR:
    current_terminal  = new ErrorTerminal(current_terminal);
    top_non_terminal = SYMBOL_error;
    goto *states.top();
```

The GOTO table entry for a state is extended to support error handling, so if the topmost non-terminal is an error it will pop a symbol and state off the stack and jump to the GOTO table entry of the topmost state. Alternatively if the state of the GOTO table entry can shift an error terminal it will jump to this shift action. Note that the default action of states that shifts error terminals is not to jump to `ERROR`, but to discard the current terminal, read a new one and try to use this.

```
Shift7:
    symbols.push(current_terminal);
    current_terminal = lexer->nextToken();
State7:
    states. push(&&Goto7);
    switch(current_terminal->token()){
        case TOKEN_minus: goto Shift3;
        case TOKEN_lp: goto Shift5;
        case TOKEN_int: goto Shift6;
        case TOKEN_error: goto Shift15;
        default: goto ERROR;
    }

Goto7:
    switch(top_non_terminal){
        case SYMBOL_E: goto State14;
        case SYMBOL_F: goto State4;
        case SYMBOL_error:
            goto Shift15;
        default: goto FATAL_ERROR;
    }
Shift15:
    symbols.push(current_terminal);
    current_terminal = lexer->nextToken();
State15:
    states. push(&&Goto15);
Switch15:
    switch(current_terminal->token()){
        case TOKEN_semicolon: goto Reduce3;
        case TOKEN_EOF: goto Reduce3;
        default:
            (( ErrorTerminal*)symbols.top())->discard(current_terminal);
            current_terminal = lexer->nextToken();
            goto Switch15;
    }
Goto15:
    switch(top_non_terminal){
        case SYMBOL_error:
            states. pop();
            symbols.pop();
            goto *states.top();
        default: goto FATAL_ERROR;
    }
```

8.7 Grammar For groo

$Groo$	::=	**newline** Top_defs	*groo*
	$\mid$	Top_defs	
Top_defs	::=	Top_def Top_defs	*Top level grouping*
	$\mid$	Top_def	
	$\mid$	**error**	
Top_def	::=	**class** id **:** **indent** $Members$ **dedent**	*Class*
$Members$	::=	$Member$ **newline** $Members$	*Member grouping*
	$\mid$	$ClosedMember$ $Members$	
	$\mid$	$Member$	
	$\mid$	$ClosedMember$	
$Member$	::=	$Types$ **id**	*Open Member declarations*
	$\mid$	$Types$ **id** **assignment** $Expr$	
$ClosedMember$	::=	$Types$ **id** $ParamsBlock$ **:** $Block$	*Closed Member declaration*
	$\mid$	$Types$ **id** **assignment** $ClosedExpr$	
$Types$	::=	$Type$ **,** $Types$	*Type grouping*
	$\mid$	$Type$	
$Type$	::=	**id**	*Types*
	$\mid$	$Type$ **arrow** $Type$	
	$\mid$	**(** $Types$ **)**	
$ParamsBlock$	::=	**(** $Params$ **)**	*ParamBlocks*
	$\mid$	**( )**	
$Params$	::=	$Param$ **,** $Params$	*Param grouping*
	$\mid$	$Param$	
$Param$	::=	$Type$ **id**	*Parameter*
$ArgsBlock$	::=	**(** $Args$ **)**	*ArgBlocks*
	$\mid$	**( )**	
$Args$	::=	Arg **,** $Args$	*Arg grouping*
	$\mid$	Arg	
Arg	::=	$Expr$	*Argument*

Table 8.4: Concrete syntax for groo.

$VarExpr$	::=	**id**	*Variables*
	|	$Expr$ **dot id**	
$Expr$	::=	**minus** $Expr$	*Open expressions*
	|	$Expr$ **div** $Expr$	
	|	$Expr$ **mul** $Expr$	
	|	$Expr$ **modulo** $Expr$	
	|	$Expr$ **minus** $Expr$	
	|	$Expr$ **plus** $Expr$	
	|	$Expr$ **shift_left** $Expr$	
	|	$Expr$ **shift_right** $Expr$	
	|	$Expr$ **greater_than_or_equal** $Expr$	
	|	$Expr$ **less_than_or_equal** $Expr$	
	|	$Expr$ **greater_than** $Expr$	
	|	$Expr$ **less_than** $Expr$	
	|	$Expr$ **equality** $Expr$	
	|	$Expr$ **inequality** $Expr$	
	|	**not** $Expr$	
	|	$Expr$ **and** $Expr$	
	|	$Expr$ **or** $Expr$	
	|	**int**	
	|	**string**	
	|	**real**	
	|	**bool**	
	|	$Expr$ $ArgsBlock$	
	|	$VarExpr$ **assignment** $Expr$	
	|	$VarExpr$	
	|	**(** $Expr$ **)**	
$ClosedExpr$	::=	$ParamsBlock$ **arrow** $Type$ **:** $Block$	*Closed expressions*
	|	$Expr$ **equality** $ClosedExpr$	
	|	$VarExpr$ **assignment** $ClosedExpr$	
$Block$	::=	**indent** $Stmts$ **dedent**	*Block structure*
	|	$Stmt$	
$Stmts$	::=	$Stmt$ **newline** $Stmts$	*Statement grouping*
	|	$ClosedStmt$ $Stmts$	
	|	$Stmt$	
	|	$ClosedStmt$	
$Stmt$	::=	$Expr$	*Open statements*
	|	**return** $Expr$	
	|	**return**	
	|	**var** $VarDecls$	

Table 8.5: Concrete syntax for groo.

ClosedStmt	::=	**if** *Expr* **:** *Block* **else:** *Block*	*Closed statements*
	|	**if** *Expr* **:** *Block*	
	|	*ClosedExpr*	
	|	**while** *Expr* **:** *Block*	
	|	**var** *ClosedVarDecls*	
ClosedVarDecls	::=	*VarDecls* **,** *ClosedVarDecl*	*Closed variable grouping*
	|	*ClosedVarDecl*	
ClosedVarDecl	::=	**id assignment** *ClosedExpr*	*Closed variable declaration*
VarDecls	::=	*VarDecls* **,** *VarDecl*	*Open variable grouping*
	|	*VarDecl*	
VarDecl	::=	**id**	*Open variable declaration*
	|	**id assignment** *Expr*	
Top_def	::=	**enum id: indent** *Labels* **dedent**	*Enum*
Labels	::=	*Label* **newline** *Labels*	*Label grouping*
	|	*Label*	
Label	::=	**id assignment int**	*Label declaration*
	|	**id**	

Table 8.6: Concrete syntax for groo continued.

groo's concrete grammar is shown in tables 8.4, 8.5, and 8.6. It is the exact grammar that is fed to the parser generator. Grammar rules of note are the rules marked "grouping", which are designed to use the `setNext()` method on the given node, thus creating a list of the given node; for instance the Members grouping. The first two grammar rules in Members have action code that tells Member to set Members as its next. Stmts and closed and open variable groupings are the only rules which do not follow this pattern, as we must ensure that we cannot have a closed variable declaration in the middle of a sequence of variable declarations. Statements are special, as a statement can be a sequence of variable declarations. Using `setNext()` here would cause the sequence of variable declarations to be lost. `append` is used here instead, running through the variable declaration list and appending the next on the last variable declaration.

We must also distinguish between an open or closed member, expression, and statement, as well as variable declaration. Open or closed refers to whether or not the given rule ends with a newline or a dedent. This is an issue, because a sequence of rules that ends on a block will not be seperated by a newline. We must therefore distinguish between a "closed" rule (a rule that ends on a block) and an "open" rule (a rule that does not end on a block, therefore, a sequence of these rules will be seperated by newlines).

Part IV

Contextual Analysis

CHAPTER 9

Contextual Analysis

Introduction

In the previous part we covered the essentials of syntactical analysis, which resulted in an abstract syntax tree, which we needed for the next step of the multi-pass compiler for groo. This step is called the contextual analysis stage, where we describe how the meaning of the program is derived and how type checking is implemented. To begin with we will introduce the general organisation of classes and techniques employed in the various steps that constitute the contextual analysis stage.

9.1 Abstract Syntax Tree (AST)

All nodes in the AST have `Node` as base class. `Node` contains a single, virtual method called `accept`, which enables operations to be performed on the tree with double-dispatching. This scheme is widely known as the visitor pattern.

Special nodes in the AST implement the `LinkedNode` template. This acts as a single linked list of nodes of the same type. For example, `MemberNode`, which is the abstract node for instance variables and instance methods. Each `MemberNode` has a `next()` that points to the next `MemberNode`, except the last one, where the end of the list is marked using `NULL`. `MemberNode` can also set its `next()`, using `setNext()` or it can `append()` a `MemberNode` to the end of a list of `MemberNode`s.

There are five major categories in the AST. Constructs, Statements, Expressions, Terminals, and Types. Constructs are top level nodes such as enums and classes, as well as the member nodes of enums and classes; fields, methods and labels. Furthermore, since parameters are primarily used by constructs, `ParamNode` is also recognised as a Construct. Construct nodes are explicitly typed; however the parser itself only provides a `SimpleType`, which is later visited to establish the proper type. Constructs also provide access to their body.

Statements make up the body of Constructs and other Statements. Among statements are if, while, variable declaration, and return as well as a statement which encapsulates an Expression. Statements do not contain type information.

Expressions represents all the arithmetic that can be performed in groo, such as addition, multiplication, bit shifting, or creating anonymous functions. All Expressions contain type information, which is set during type checking.

Terminals are special nodes, which are used to encapsulate data from the code, such as strings, integers, or identifiers of variables or types.

Finally there are Types, which are nodes that denote a type. Certain types; `InferredType`, `TupleType`, and `SimpleType` are utility types, and do not denote a type in themselves. They do however point to another type, which eventually will point to a proper type.

Some Statements and almost all Constructs are sub classes of `VariableDeclaration`. These nodes contain extra information used in codegeneration, which is not attended until the Allocation visitor.

9.2 Visitors

The visitor pattern allows easy traversal of the AST during the various stages of compilation. For example, when creating a graphical representation of the AST, the DotBuilder visitor navigates the AST, visiting each node; drawing the tree as it goes.

By using this pattern it is possible to create an arbitrary number of visitors. We have implemented the following visitors:

DotBuilder Generates a graphical representation of the AST. Useful for debugging.

NodePrinter Outputs the AST to the console in a readable format. Also useful for debugging.

ErrorFinder Reports syntactical errors, including line number and token position.

TypeChecker Performs contextual analysis, which includes type checking.

Intepreter Semantics driven recursive interpreter.

AllocationVisitor Calculates the amount of memory needed to be allocated before running Codegen.

Codegen Compiles the AST into the intermidiate language gril.

VROOM Iterative interpreter.

It would be relatively easy to implement other useful visitors, such as a pretty printer, which formats code, or a spell-checker.

Visitors inherit from the `Visitor` class. The base class contains overloads of the virtual `visit` method for each node represented in the AST.

As mentioned earlier, the visitor pattern employs the concept of double-dispatching, which is useful as it adds a level of abstraction to solve the problem of calling the correct concrete `visit` method for a concrete node object. This is done by dispatching the method call depending on the runtime type of a given node.

In this way we avoid having to perform explicit type checking to invoke the correct concrete method. In our implementation, each concrete `visit` method is distinguished by its formal parameter. An alternative naming convention would be to denote each method as `visitConcreteNodeX(ConcreteNodeX node)`, however we decided to make use of the method overloading available in C++, since the formal parameter provides enough clarity. An informal illustration of the double-dispatch performed in the visitor pattern is shown in figure **??**.

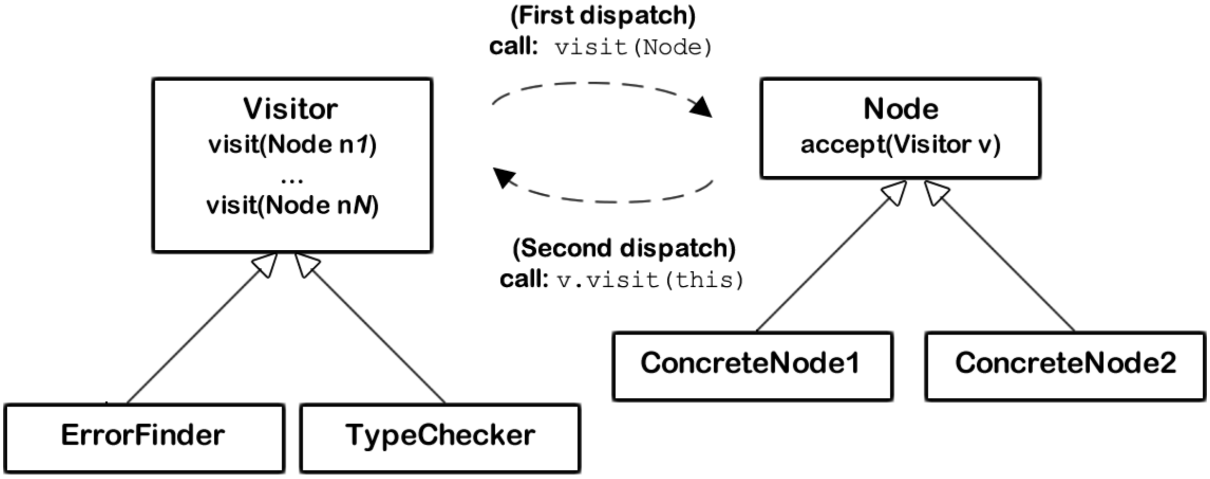


Figure 9.1: The visitor pattern.

The different visitors may traverse the AST systematically with depth-first, breadth-first or a hybrid search algorithm.

A depth-first traversal works by visiting nodes in a first-in-last-out manner, the ordering is usually maintained by a stack.

Breadth-first traversal is performed on nodes first-in-first-out, with a queue-like data structure, so the AST is visited in a sideways manner.

For instance, the DotBuilder performs a depth-first traversal of the AST. The Type-Checker uses a hybrid of the two searches. It uses breadth-first search in order to gather knowledge of classes, instance variables, methods, and enumerations, after which it switches to depth-first traversal in method bodies.

9.3 Type Checker

The `TypeChecker` visitor is responsible for ensuring that every variable is declared before use and that the program is well-typed.

There are two types of declarations - `VariableDeclarations` and `TypeDeclarations`. A `VariableDeclaration` associates a `TypeDenoter` to a variable in the `IdTable`. A `TypeDeclaration` binds a `TypeDenoter` to an `Identifier`, indicating that the `Identifier` represents the type. The following nodes are declarations:

ClassNode

A `ClassNode` is a `VariableDeclaration` as it acts as a constructor. A `ClassNode` is also a `TypeDeclaration` as a class is a type in groo.

EnumNode

An `EnumNode` is a `TypeDeclaration`, as an enum is a type in groo.

FieldNode

A `FieldNode` is an instance variable - therefore it is a `VariableDeclaration`.

MethodNode
Method names are treated as variables in groo.

PrimitiveDeclaration
`PrimitiveDeclaration` is a `TypeDeclaration` for primitives.

VarDeclStmt
`VarDeclStmt` is used for variable declarations inside a method, and is therefore a `VariableDeclaration`.

A `TypeDenoter` is an abstract class, which denotes a type. There are several different kinds of `TypeDenoters`, each representing their own type. Some are also for utility purposes, such as grouping types in a tuple. The following classes are type specialisations:

ClassType
Contains the declaration of the class.

EnumType
Contains the declaration of the enum.

FunctionType
Contains two `TypeDenoters` - the type of the parameters and the return type. These types can be any of the other types.

InferredType
Special type which has no initial type. It assumes the first type it is compared to. Mainly used as part of a `FunctionType`.

PrimitiveType
Represents the primitive types. Implements the flyweight pattern.

SimpleType
Is used by the parser, to represent explicitly written types. When visited it performs a lookup on the identifier, and points to the `TypeDeclaration`.

TupleType
Is used to create a tuple of types. Has a `next()` and a `TypeDenoter`.

`TypeDenoter` implements `isSubType()` and `isSuperType()`, using double dispatch. The base class returns $false$ by default allowing each specific type to only implement those who can return true. `InferredType`, `SimpleType`, and `TupleType` are an exception to this. These types are "utility" types, and the actual type comparison is performed on the type they hold. `isSubType()` is used for comparing types, allowing inheritance. For types which have no inheritance, it checks if they are equal. When calling `isSubType()`, it calls `isSuperType()` of the specific type, allowing us to easily add new types by overloading `isSuperType()`.

Before the `TypeChecker` begins traversing the AST, all explicitly written types (such as the return and parameter types of `MethodNodes`) must be visited by the

IdVisitor. These types must be visited, because the parser creates them as SimpleTypes. SimpleTypes declaration is initially set to NULL. They must therefore be visited, so a lookup can be performed on the id and the declaration can be set. While the IdVisitor visits these nodes, it also adds the TypeDefinition of classes and enums to the scope, sets up the standard environment, and decorates classes with the declaration of their parent, if they have a parent.

9.3.1 Type checking

Once the IdVisitor completes its short traversal of the AST, the TypeChecker is ready to begin.

Top level type checking

When the TypeChecker traverses the AST, it begins as a breadth-first traversal, until it has visited all ClassNodes and EnumNodes. These two nodes must be visited before members are visited, as their VariableDeclaration must be added to the top-level scope in the event that a member, or body of a member requires their functionality. After all ClassNodes and EnumNodes have been visited, the TypeChecker begins visiting class members. Each ClassNode enters a new scope when they begin visiting members.

Type checking members

FieldNodes must check if their id has previously been declared. If this is the case, they report an error. Otherwise they add their declaration to the scope.

Classes can override their parent's methods if the method signature matches the parent class method. Therefore, if their id already exists in the scope, they must ensure that their signature is the same as the parent method. If this is not the case, they report an error. If there is no overriding, the MethodNode adds its declaration to the scope. They then visit their siblings.

Once the member siblings have been visited FieldNode and MethodNode perform differently. For FieldNodes the type checker performs type checking on its expression, if it has one. This is done by visiting the expression. Ultimately, visiting an expression will update the expressions type. How this is done varies from expressions. A more interesting expression, binary operations, will be detailed further down. Once the expression has been visted, isSubType is called on the expression type, with the FieldNodes declaredVariableType as the parameter, and reports an error if the check fails.

MethodNodes are more interesting. First, the type checker enters a new scope, giving the scope the expected return type of the method. Then, the type checker begins visiting the parameters on the MethodNode if any. Visiting a ParamNode ensures that the parameter variable does not exist in the scope, add its VariableDeclaration, and then visit the next parameter. With parameters added to the scope, the type checker begins visiting the code body of the method, which is comprised of StmtNodes.

Type checking statements

Visiting an IfStmt or a WhileStmt is similiar. The type checker ensures that their conditional expression is of type bool, enters a new scope, and begin type checking their code bodies. ExprStmt is simply a place holder for expressions, so the type checker visits the expression.

Expressions of ReturnStmts can be null. In this case, the type checker creates a void type comparison. Comparing return types is done in the IdTable, since the expected return type is held in its respective scope. IdTable will look upwards through the scope until it finds a scope with a defined return type. The actual comparison is done by ensuring that the expression is a subtype of the return type, using the same method as the type checker to compare types.

The type checker will also ensure that a method ends on a return, or allow a void method to return implicitly, by using a macro for visiting the next. The type checker will also warn users of dead code, if ReturnStmt has a next value, indicating that there are statements after the return, which cannot be accessed.

VarDeclStmt is the statement that declares variables. Assigning them immediately is optional, and the variable then may not have a type. If there is no type assigned to a variable, they will return NULL, which, when compared using isSubType, will always return false. The type checker also checks that the variable has not been declared previously in the scope, or in any higher level scopes.

Type checking expressions

Visiting expressions will always result in visiting one of the four primitive literals, int, float, bool or string, or by visiting the VariableExpr. The four literals will set the type to the type they each represent, whereas VariableExpr must visit its Variable to get the VariableDeclaration. Once it has the VariableDeclaration it can set its type to VariableDeclaration's declaredReadOnlyVariableType. It is necessary to look at the declaredReadOnlyVariableType, because all VariableDeclarations implement it. Not all VariableDeclarations implement declaredVariableType. An example of one such VariableDeclaration is the ClassNode. This indicates that the ClassNode's identity cannot change type e.g. you cannot assign a new value to the class name.

Before this of course it must visit its Variable. There are two different kinds of Variables. The first is the simple Variable, which contains an Identifier. When the type checker visits this Variable it must lookup the declaration in the IdTable, and then set the Variable's declaration to the lookups result. The second Variable is a specialisation of Variable called AttributeAccessVariable. The slightly over complicated name implies that this variable is not inside our current or parent scopes. To access it the AttributeAccessVariable has provided an attribute that the type checker must visit. The attribute itself is an expression and could therefore be anything from a Variable to another AttributeAccessVariable or even a CallExpr. Once the attribute has been visited, the type checker performs a getAttribute call on the attribute's type. getAttribute is very similar to the lookup function on the IdTable. Instead of looking for the VariableDeclaration in the IdTable, it will look in the current type. The different TypeDenoters implement their version of getAttribute. ClassType's will look through their class declaration for members matching the variable, EnumNode's look through their labels and so forth.

Another expression is the BinaryOperatorExpr. In groo, all binary operations are handled like instance methods of the left hand side expression. Expression 9.1 would therefore be treated like expression 9.2 in groo.

$$Expr1 + Expr2 \tag{9.1}$$

$$Expr1.op\_add(Expr2) \tag{9.2}$$

To get the type information of the op_add method, the type checker must first perform a lookup on the operator, to get an id for the method. The type checker then calls getAttribute on the first expression's type. This returns a VariableDeclaration which can be used to get the TypeDenoter of the declaration. With the method type secure, the type checker must ensure that the "method call" is correct, and it constructs a FunctionType for comparison, with the type of the second expression as the parameter. Because the binary operation itself gives no indication of the return type, the constructed FunctionType is given InferredType as the return type, so it will automatically assume the return type of the acquired declaration.

This also means that if operator overloading was implemented in the future, the getAttribute method could be used to get information about the operator type, when performing operations on non primitive types, or when the primitive type operators have been overloaded.

9.3.2 Identification Table

The TypeChecker makes use of an identification table, called IdTable, which helps keep track of both type and variable declarations in their respective scopes. When the TypeChecker starts, all of the basic primitives, such as bool, float, int, string, and void are added to the global scope. This way the primitive types will always be declared by default before type checking is performed.

This makes it relatively easy to plug new primitives into the standard environment, should the need arise.

In addition, IdTable contains a class called Scope, which basically is a linked list. Each Scope has a pointer to its parent. When enterScope() is called, a new scope is attached to the linked list. Leaving the Scope is done by calling leaveScope(), which removes the current Scope and sets its parent to be the current Scope. A lookup on a variable or type declaration is performed at the current scope and then recursively 'upwards' by following the pointer to the parent Scope. Variable and type declarations are stored in separate dictionaries. So, the insert() method is overloaded for either case. This enables the programmer to declare variables with type names, for instance the declaration of var int = 0 is possible.

Part V

Execution

CHAPTER 10

Interpretation

When a program has gone through contextual analysis without errors it is ready to be executed. groo can currently be executed using two different techniques. The recursive interpreter uses the AST to execute groo. And the virtual machine VROOM, covered in the next chapter, uses the intermediate language, gril, which is translated from a groo program by the code generator.

Interpreting a source program is to execute it immediately without first translating it to low-level instruction code [Watt and Brown, 2000].

We have chosen to focus on two different approaches to interpretation: iterative and recursive interpretation.

Iterative Interpretation

Iterative interpretation is used to execute a program consisting solely of simple instructions. It is a simple fetch-analyse-execute cycle, where a series of instructions are fetched, analysed, and then executed [Watt and Brown, 2000].

Iterative interpretation with a virtual machine is explained in section 11.

Recursive interpretation

Recursive interpretation is needed to interpret more complex instructions - such as statements and expressions. In contrast to iterative interpretation, recursive interpretation works in two steps. The first step is parsing and analysing the program. The second step is to recursively execute the program. The execution step can be implemented using the visitor pattern, traversing the decorated abstract syntax tree. Recursive interpretation often results in slow execution speed. This is one of the reasons why recursive interpretation is not widely used in higher level programming languages [Watt and Brown, 2000].

Implementing a recursive interpreter is a quick way of testing a programming language.

10.1 groo Recursive Interpretor (gri)

gri is a semantics driven recursive interpreter. This means that we will use the same concepts as in the operational semantics given in chapter 5.

Variables are saved in an environment, which has a pointer to a location where the value is stored.

The sections below will describe the memory storage and how some of the more complex nodes are interpreted.

Variable Storage Allocation

The class `loc` is the primary storage class. It contains an instance variable of the base class `Value`. The classes `ObjectValue`, `FunctionValue`, `ConstructorValue`, `IntegerValue`, `FloatValue`, `BooleanValue`, and `StringValue` are all specialisations of `Value`. This will allow `loc` to hold a pointer to the location of all variables, objects and functions in memory.

`env` is the environment in which one or more `loc`s can be stored. `env` is essentially a linked list where each node contains a `loc` and an `Identifier`. A look up can be performed in an `env` and the location of the `Identifier` will be returned. An example of the association between `env`, `loc`, and `Values` can be seen in figure **??**.

Interpreting a groo program begins with an empty `env`. All `ClassNode`s are then visited and stored in the `env`. The interpreter holds a number of instance variables used to store and return values when visiting the AST.

- `env`: This variable holds the environment of the scope the interpreter is currently in.
- `loc`: Variable used to return a location.
- `retenv`: Variable used to return an environment `env`.
- `result`: Variable used to return the `Value` of a return statement.
- `retval`: Variable used to return a `Value`.

Memory Management

The semantics for groo does not specify how memory should be deallocated. gri uses reference counting to determine when memory can be released. This approach have been chosen because it is simple to implement. The disadvantage is the overhead of incrementing and decrementing the reference counter, however, recursive interpretation is already slow. Furthermore, reference counting cannot release memory if it contains cycles in the references. [Aho et al., 2006, section 7.5.3]

All `Value` will be given a reference counter, when a `loc` is assigned a new `Value`, it will decremented the counter of the old `Value`, and increment the counter of the new `Value`. If a counter reaches zero, no `loc` is currently pointing to that `Value` and it can be safely deleted.

When the program exits a scope all variables created in that scope are discarded. In order to delete the `loc`s and `Value`s that are no longer in use `env`s have a reference counter referring to each element in its list; every time `env` is updated, the reference counter is incremented. If a scope is entered, a copy of the environment `env`, with an incremented reference counter is saved in a temporary environment `env1`. When the program leaves the scope `env` will decrement its reference counter. This will cause the element `env` is referring to, to be removed, decrementing the reference counter of the element pointing to the now nil-element. This causes all new entries in the environment to be recursively deleted, and thus `env` must again refer to `env1`.

If the reference counter reaches zero the env is deleted along with its loc. The next env in the list then removes the reference to the deleted env and its reference counter is decremented. If a loc is deleted the Value is no longer referred to, and its reference counter is decremented, and is deleted if it reaches zero.

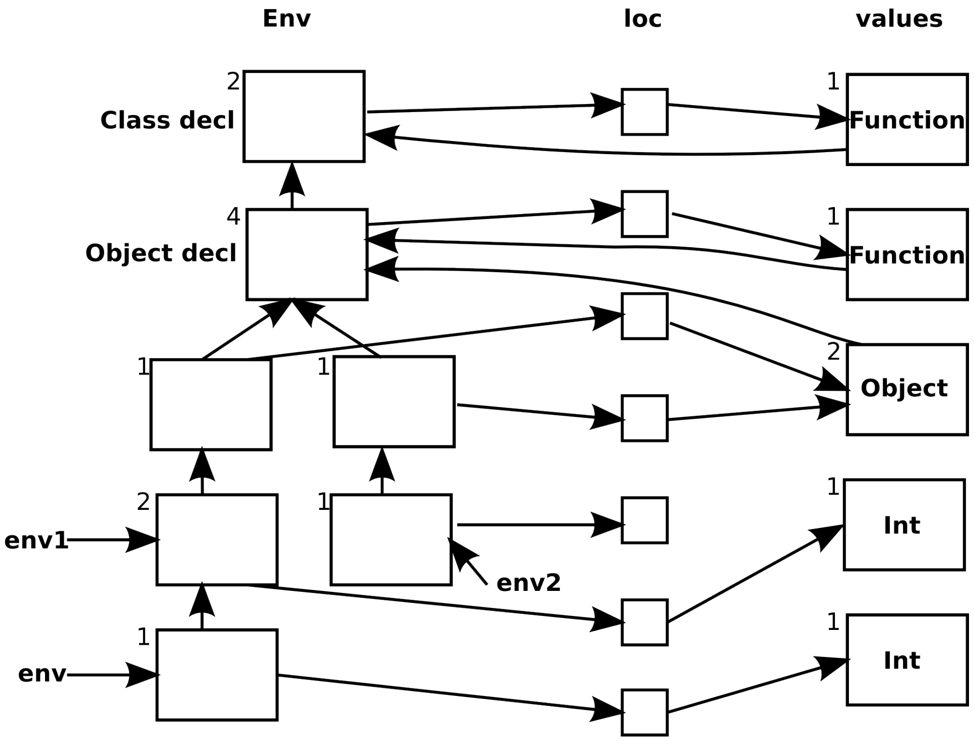


Figure 10.1: An example of the associations between environment, location and variables. The number in the upper left corner denotes the reference counter.

Class Declarations

Classes are forward-declared by updating env to point to new locations for each class. After each class has been declared, a new constructor value is created containing the class members and the env.

Class Members

Class members are fields and methods. env is updated with the name of the member pointing to a new location. As members are also forward declared, env is updated for each member before creating the values.

Fields can be assigned an expression, and if so, the expression is accepted and the return value is stored in the location of the field's id.

The value of a method is a function value, containing the body and parameters of the method and the current env.

Statements

Statements can either declare new variables or change the values of existing variables. If- and while-statements evaluate their condition and execute the corresponding branch. The resulting `env` is then saved, so that any changes made in the body of the if- and while-statements can be discarded afterwards.

All statements expect the return statement to have a `next`-pointer to the next statement, and this statement is evaluated at the end of the current statement's execution.

Variable Declarations

Variable declarations will declare new variables. `env` is thus updated with a new location for the id of the variable. A value is given to this location, if an expression has been assigned to the variable. If so, the expression is evaluated, and the value is retrieved from `retval`.

Expressions

Expressions can only change values of already existing variables. This can either simply be by assigning the value of an existing variable, or by performing operations on variables or literal values. The result of evaluating an expression is saved in `retval`.

For the simple binary and unary operators, the expressions are evaluated, and the apply-function is called using the operator as an argument. The apply-function implements binary and unary operations for primitive types.

Call expressions will have to find the class function, evaluate the body, and then return the result. Because the body can contain statements that can declare new variables, we save a local version of the `env` before evaluating anything else. The function to call is found by evaluating the callee of the expression. If this is a `VT_FunctionValue` it is a call to a function or method. On the other hand, if it is a `VT_ConstructorValue` it is a call to a constructor.

For a call to a function or method we find the `env` of the function, which contains the variables accessible from within the function. The arguments and parameters are then evaluated and declared in the `env` of the function. A new location is created for the parameter, and the value found by evaluating the argument's expression is saved at that location. After all the arguments and parameters have been declared in this `env`, the body of the function is executed in it. The result, found in `result` is saved in `retval`.

If the call expression was a constructor we would evaluate the body of the constructor in the `env` of the constructor, as with functions and methods. The return value is then an `ObjectValue` with the `env` of the constructor, containing the members of the object.

A local version of `env` saved at the beginning is set to the working `env` again, to discard all changes that might have been made.

Variable Expressions

Variable expressions are used to find the locations of variables in `env`. The location of a simple variable is found by looking up its id in the current `env`. But a variable expression can also be a sequence of calls to methods and fields. Such a variable expression has an expression for the first part, and ends in an id. The location is found by accepting this expression, and thereby updating `retval` to contain an `env`, which is an object value. The location can then be found by looking up the remaining part of the variable expression, id, in this `env`.

Literals

Literals are either integers, floats, bools or strings. A new value of with the corresponding type is created from the value of the literal, and saved in `retval` where it can be accessed by the called expression.

CHAPTER 11

VROOM, Virtual groo Machine

11.1 A Virtual Machine for Groo

A brief discussion of the advantages and disadvantages of a virtual machine.

A virtual machine is a software implementation of a machine executing an intermediate language. Compared to the recursive interpreter a virtual machine may run almost ten times faster [Watt and Brown, 2000]. When a program runs on the virtual machine, the machine-specific architecture is abstracted away, so the semantics of the program remain the same, regardless of what hardware it is running on.

This means that instead of rewriting and compiling a program inorder to port it to a new platform, a virtual machine for the new platform is the only thing that needs to be ported. In the early development phases of a programming language it can be an advantage to compile the code to an intermediate language, rather than directly to machine code, as this requires much less work. In some cases a suitable virtual machine may already exist. This saves the time it takes to create the intermediate language and interpreter from scratch [Watt and Brown, 2000].

The primary reason for creating a virtual machine for groo is to be able to execute groo programs more efficiently. With a proper intermediate language this can be done by using an iterative interpreter, rather than a recursive interpreter, as seen earlier.

Figure 11.1 illustrates how a groo program may be translated to the intermediate *gril* language (**GR**oo **I**ntermediate **L**anguage) and interpreted on the machine, *M*.

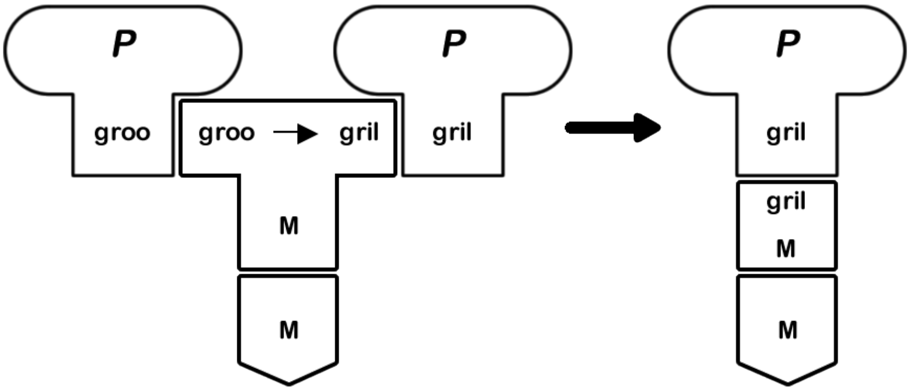


Figure 11.1: A tombstone diagram displaying the compilation and interpretation of the groo program, *P*.

11.2 Runtime Organisation in VROOM

In this section it is described how a groo program is organised at runtime within the virtual machine. VROOM is a stack machine, which means that variables are pushed and popped of a stack in almost every instruction. In order to easily support closures, used for higher-order and anonymous functions, we have decided to allocate stack-frames (from now on referred to as frames) on the heap as a linked list of frames. Though, we will also have an expression stack for evaluating expressions, so that memory for temporary variables need not be allocated on the frames. Therefore the frames can be allocated with a constant size.

VROOM uses an accurate tracing garbage collector to manage memory. To facilitate this feature VROOM distinguishes between reference types and value types. This results in two expression stacks (stacks used for evaluation expressions), a stack for reference types and a stack for value types. Frames are also divided into two sections, a section where references types may be stored and a section where value types may be stored.

In the implementation this is done by letting a frame pointer point into the frame where the two sections meet. Then reference types can be located by subtracting their offset from the frame pointer, and value types can be found by adding their offset to the frame pointer. This is illustrated in figure 11.2, which gives an impression of how things are organized at runtime. Hexadecimal numbers are pointers, and the arrows indicate what they are pointing to. Notice in figure 11.2 that the vtable pointer on a object points to a segment of literals in the code store, and that this pointer is considered a value type; this ensures that it will not be subject to garbage collection.

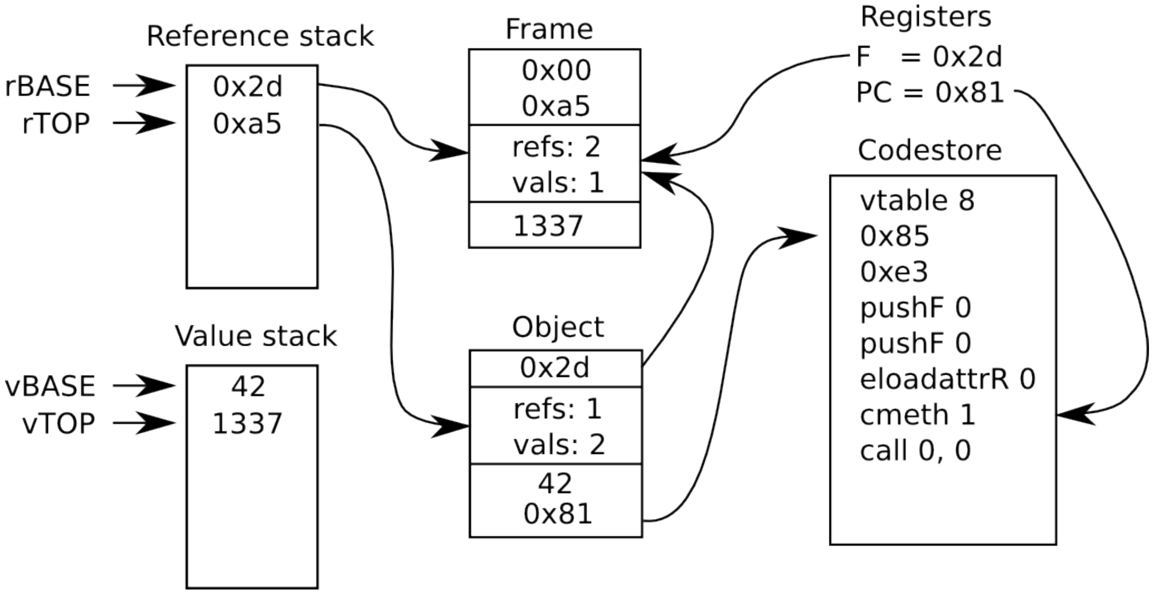


Figure 11.2: Organization of memory at runtime within VROOM.

11.2.1 Registers

VROOM is a stack machine with few registers. These registers are implicitly encoded in the OpCode and are never specified explicitly. In table 11.1 the registers are listed.

Table 11.1: Registers in VROOM

Register	Type	Description
`PC`	CodePoint	The program counter, which points to the next instruction.
`PBASE`	CodePoint	Program base. This register remains constant and is used as a pointer-offset for vtables.
`F`	Frame*	The current frame.
`rCBASE`	Ref*	The call base on the reference stack, used to return from function calls and load arguments.
`vCBASE`	Val*	The call base on the primitive stack, used to return from function calls and load arguments
`rTOP`	Ref*	A pointer to the reference at the top of the reference stack.
`vTOP`	Val*	A pointer to the value at the top of the primitive stack.
`rBASE`	Ref*	Base of the reference stack.
`vBASE`	Val*	Base of the primitive stack.

11.3 gril, groo Intermediate Language

Description of the groo intermediate language.

11.3.1 Instruction layout

An instruction is made up of an 8-bit operation code and 0-32 bit argument. There are several ways of encoding arguments, depending on the operation code (OpCode). Some instructions are followed by literals, where the argument specifies how large the literal data following the instruction is.

$$\underset{\text{8 bit}}{\underline{[opcode]}}\;\underset{\text{0 bit - 32 bit}}{\underline{[arguments]}}$$

Since an instruction can be mixed and matched with several combinations, depending on its purpose, we will elaborate on this in the following listing.

OpCode + CodePoint Typically used for a jump or conditional jump to a specific address in the code store. These instructions have a length of 40 bits.

OpCode + Offset Typically used for loading or storing data from/to frame or object. These instructions have a length of 24 bits.

Opcode + Depth Used for instructions that need to push a frame. Depth tells us how many enclosing frames we must travel up in order to locate the desired frame. These instructions have a length of 16 bits.

OpCode + Size Used for instructions that indicates that there is a literal in the code store. These instructions have a length of 24 bits + the literal data that follows.

OpCode + Refs + Vals Generally used for instructions that allocate memory, here Refs and Vals tell us how many references and values the object allocated must hold. These instructions have a length of 40 bits.

11.3.2 Instruction Set

Table 11.2 lists the basic gril instructions. Instructions corresponding to the unary and binary operators are listed in table 11.3. Opcode arguments have been abbreviated in the table listings. An example of a small groo program and its gril representation, with a detailed explanation of how the execution is performed can be found in appendix B.2.

Table 11.2: Basic Instructions. Argument abbreviations: cp = 32bit codepoint, s = 16bit size, d = 8bit depth and o = 16 bit offset, (r,v) = 16 bit number of reference type and 16 bit number of value types.

Opcode	**Args**	**Description**
`jump`	cp	Sets the program counter to the specified code point.
`condjump`	cp	Jumps to cp if the value on the top of the stack is 1.
`returnR`	(r, v)	Return a reference type as result from a function that was given r reference and v value type arguments.
`returnV`	(r, v)	Return a value type as result from a function that was given r reference and v value type arguments.
`call`	(r, v)	Call a function with r, v arguments, reference and value types respectively.
`frame`	(r, v)	Set F to a new frame f with space for r reference and v value types, set current F as parent for f.
`pop`		Set F to the parent of current F.
`epopR`		Pop reference of the reference stack
`epopV`		Pop value of the value stack
`pushF`	d	Pushes the current frame onto the stack. If $d > 0$, follow the frame's parent chain.
`eloadattrR`	o	Take topmost reference of the reference stack, and push its o reference attribute on to the reference stack
`eloadattrV`	o	Take topmost reference of the reference stack, and push its o value attribute on to the value stack
`estoreR`	o	Store the topmost reference on the reference stack to the second topmost reference on the reference stack with offset o.
`estoreV`	o	Store the topmost value on the value stack to the topmost reference on the reference stack with offset o.
`cmeth`	o	Create function from o offset in the vtable in of the topmost reference on the reference stack.
`cfunc`	cp	Pushes a closure onto the reference stack with a pointer to the top frame and the code point, cp.
`argload`	(r, v)	Loads r, v arguments, reference and value types respectively into the current frame.
`halt`		Stop execution with topmost value on the value stack as exit code.
`new`	(r, v)	Push new object, with space for r, v values of reference and value types respectively, on to the reference stack.
`loadl`	s	Load s literals from code store following this instruction onto the value stack.
`vtable`	s	Indicates a vtable of size s in the code store, the virtual machine should exhibit undefined behaviour if this instruction is executed.
`nop`		No operation, this instruction is skipped.
`pushNULL`		Push 0 onto the reference stack.
`dupR`		Duplicate top of reference stack.
`dupV`		Duplicate top of value stack.

Table 11.3: **Operator instructions** works by taking the two topmost values from the value stack and operating on these as specified, pushing the result on to the value stack. Instructions postfixed with capital R takes the values from the reference stack.

Opcode	Meaning
`iadd`	$v' := i_1 + i_2$
`isub`	$v' := i_1 - i_2$
`imul`	$v' := i_1 \cdot i_2$
`idiv`	$v' := i_1 / i_2$
`shiftl`	$v' := i_1 \ll i_2$
`shiftr`	$v' := i_1 \gg i_2$
`mod`	$v' := i_1 \bmod i_2$
`ieq`	$v' := i_1 = i_2$
`ineq`	$v' := i_1 \neq i_2$
`ilt`	$v' := i_1 < i_2$
`ile`	$v' := i_1 \leq i_2$
`igt`	$v' := i_1 > i_2$
`ige`	$v' := i_1 \geq i_2$
`fadd`	$v' := f_1 + f_2$
`fsub`	$v' := f_1 - f_2$
`fmul`	$v' := f_1 \cdot f_2$
`fdiv`	$v' := f_1 / f_2$
`feq`	$v' := f_1 = f_2$
`fneq`	$v' := f_1 \neq f_2$
`flt`	$v' := f_1 < f_2$
`fle`	$v' := f_1 \leq f_2$
`fgt`	$v' := f_1 > f_2$
`fge`	$v' := f_1 \geq f_2$
`and`	$v' := v_1 \wedge v_2$
`or`	$v' := v_1 \vee v_2$
`eq`	$v' := v_1 = v_2$
`neq`	$v' := v_1 \neq v_2$
`eqR`	$v' := r_1 = r_2$
`neqR`	$v' := r_1 \neq r_2$

11.4 MOM: Mark-sweep Object Manager

MOM is an accurate mark-sweep garbage collector for VROOM. We have chosen to implement a garbage collector for VROOM, to demonstrate that groo can be implemented with both reference counting and garbage collection. It also worth mentioning that garbage collection is also likely to perform better than reference counting, and that garbage collection does not leak cycles, whereas reference counting does.

On the other hand, garbage collection does not exhibit the same locality as reference counting. The cost of reference counting is spread evenly throughout the execution of a program, whereas a mark-sweep garbage collector stops the execution while cleaning up. It should be noted that reference counting may cause more overhead than garbage collection, however, this depends on the behaviour exhibited by a program. A mark-sweep garbage collector causes overhead when it cleans up, while reference counting causes overhead whenever a reference type is used, e.g. passed as parameter, used in assignment, etc.

MOM is an accurate garbage collector, which means that at runtime it can distinguish between pointers and other data such as integers, which should not be considered pointers. Section 11.1 explains how reference types and value types are distinguished in VROOM.

11.4.1 Mark-Sweep Garbage Collection

MOM is a mark-sweep garbage collector which means that it performs clean-up in two phases. First it marks every reachable heap allocated structure, which we shall call heapstruct, then it releases every unreachable heapstruct [Aho et al., 2006, section 7.6.1]. Following definition 6 an unreachable heapstruct cannot be accessed by the program again. Thus, unreachable heapstructs can be safely deallocated.

> **Definition 6** A heapstruct is reachable in VROOM if it is directly or indirectly reachable from either the reference stack or the F register.

Figure 11.3 shows a pointer to a heapstruct and the metadata within the heapstruct. Whenever MOM allocates a heapstruct it sets the `reachable` bit to the global `TrueMeansReachable` bit, and sets the fields `refs` and `vals` to the number of references and values for which space has been allocated. MOM has a linked list of allocations called `allocations`. When an allocation is performed, its `next` field is set to `allocations` and `allocations` is set to a pointer to the newly allocated heapstruct. This way a linked list of all allocations is maintained.

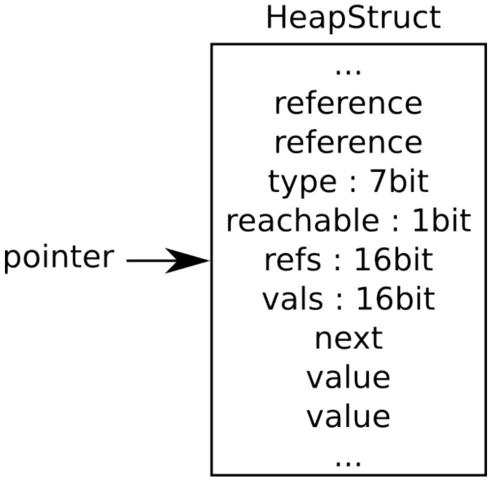


Figure 11.3: A heap allocated structure with available metadata.

When MOM is called to clean up, it flips the global `TrueMeansReachable` bit and traces the reference stack and the F register. This means that it observes each pointer that is not 0, and if the `reachable` bit of the heapstruct it points to is not equal to `TrueMeansReachable` it flips this bit, and traces the references held by the heapstruct. By following figure 11.3 the number of references held by a heapstruct and a pointer to the first of these can easily be found.

When MOM has marked all the reachable heapstructs as described above, it performs the sweep phase. Here MOM iterates through the list of heapstruct `allocations`, while removing and releasing any unreachable heapstruct, i.e. a heapstruct where the `reachable` bit is not equal to `TrueMeansReachable`.

MOM is called to clean up when a certain number of instructions has been executed. While MOM is cleaning up, the iterative interpretation in VROOM pauses.

11.4.2 Future Optimisations

The current implementation of MOM obtains memory from `calloc` and releases it using `free`. These calls are not very efficient given the number of memory allocations used in VROOM and gril programs. Faster memory allocation may improve performance significantly. One way of improving the allocation and deallocation performance would be to use free-lists, inspired by Boehm and Weiser [1988].

This could be done by allocating a large chuck of memory, m, and then split m into a linked list, f, of fixed size memory blocks. Then whenever allocating a block of memory smaller than or equal to the block size of f, a block from f can be removed from f and returned. Whenever a block of memory is released; check if the block is within m and if so, zero it and return it to f, instead of calling `free`.

This concept could be generalised and more than one free-list could be created, whilst `calloc` could still be used for large arbitrary allocations. The free-lists might also be improved even further using bit-fields to indicate which allocations are free and used. Then a bit-field could also be used to hold the reachable bit for free-list allocations during the mark phase, allowing `reachable` to be removed from the `allocations` list.

This is likely to improve performance considerably because it will reduce the entries in the list of allocations and reduce the time needed to allocate memory. Due to the way all stack-frames are heap allocated, these improvements will improve the performance for small allocations, which are used often in VROOM.

11.5 Code Templates

Code templates can be used as a tool to translate high-level code into an intermediary format, without introducing too many specifics about the actual implementation. Many of the opcodes are used in the templates and some are abstracted away. We introduce abstract operations, such as *declare*, *evaluate* and *execute* in the code templates. This allows us to keep code templates relatively simple, since the immediate constituents have their own code templates. Note that these templates do not strictly follow the abstract syntax, as we use code templates to assist and document the implementation, rather than formalizing the behaviour. Nevertheless, it could be interesting to formalize these code templates and perhaps use them to show equivalence with the semantics.

Some code templates require unknown forward jumps to labels or addresses to be declared. This is solved by allocating a slot in the code store and storing a pointer to that location. When the address is known, the operation can be patched with the correct label or address. We introduce four auxiliary functions, $refs(x)$, $vals(x)$, $size(x)$ and $\mathit{offset}(x)$. $size(x)$ returns the size of a vtable for the methods in a class. The function $refs(x)$ returns the number of reference type variables in $params$, $args$, $members$ or $stmt$, used for allocating frames and objects. The function $vals(x)$ is similar to $refs(x)$ except it counts the number of value type variables. The function $\mathit{offset}(x)$ gets the appropriate offset for instance variables and methods.

In this section we have listed selected code templates for each syntactical category.

template 1 Groo Program

```
execute ⟦ decls ⟧ =
   $\bar{l}$:  $vt$
  $\bar{ss}$:  jump $\bar{s}$
       vtable $size(decls)$
  $\bar{vt}$:  $c_1 ... c_n$
   $\bar{s}$:  pushf $0$
  $\bar{mc}$:  cmeth $\mathit{offset}(main)$
       call $0, 0$
  $\bar{mm}$:  cmeth $\mathit{offset}(main)$
       call $0, 0$
       halt
       declare ⟦decls⟧
```

Code template 1 is the first code template to be executed. The first instruction to be emitted is the vtable location, vt, this is vtable that contains all constructors. Then the vtable is skipped by jumping to $\bar{s}$, where the constructor for *MainClass* is called and the method *main* is executed. The label $\bar{vt}$ points to the constructors, $c_1...c_n$, for each declared class. This allows constructors to be called globally.

After the `halt` instruction has been emitted, the declarations are evaulated and added to the code store. In the actual implementation we save $\bar{mc}$ and $\bar{mm}$ for later, in order to patch `cmeth` instructions with the correct code point offsets while visiting class declarations. In the code template, however, this information is provided by the $\mathit{offset}(x)$ function.

Declarations

template 2 Class Declaration

```
declare ⟦class id: members ; decls⟧ =
        vtable $size(vt)$
$\bar{vt}$:    $m_1 ... m_n$
 $\bar{c}$:    loadl 1
        $\bar{vt}$
        new $refs(members), vals(members)$
        evaluate ⟦$\forall$ $type\ id$ = $e \in members$⟧
        return 0
        declare ⟦$\forall$ $type\ id$ ( $param$ ): $S \in members$⟧
```

In template 2 the first label $\bar{vt}$ is the vtable for the class, or rather code points for each method. First fields are evaluated, then methods are evaluated.

template 3 Field Declaration

```
evaluate ⟦$type\ id$ = $e$⟧ =
   dupR
   evaluate $e$
   estoreV $offset(id)$
```

Template 3 shows how instance variables are declared. Here, $offset(x)$ locates the correct field. If the field is declared without a value no action is performed in the code template. If the expression e evaluates to a reference type the operation `estoreR` is used instead of `estoreV`.

template 4 Method Declaration

```
declare ⟦$type\ id$ ( $params$ ): $S$⟧ =
   frame $refs(S), vals(S)$
   argload $refs(params), vals(params)$
   execute $S$
```

Methods are declared as shown in template 4. The instruction argload loads the arguments from call base into the current frame. The offsets are implicitly provided because parameters are always the first variables in a frame.

Statements

template 5 While Statement

```
execute $[\![$**while** $e$ **:** $S_1$; $S_2]\!]$ =
$\bar{j}$:  jump $\bar{h}$
$\bar{g}$:  frame $refs(S_1), vals(S_1)$
    execute $[\![S_1]\!]$
    pop
$\bar{h}$:  evaluate $[\![e]\!]$
    condjump $\bar{g}$
    execute $[\![S_2]\!]$
```

Code template 5 for the while statement is devised in such a way that the condition is evaluated by jumping to the $\bar{h}$ label. If the condition evaluates to $true$, the code starting at label $\bar{g}$ is executed.

template 6 If-Else Statement

```
execute$[\![$**if** $e$ **:** $S_1$ **else :** $S_2$; $S_3$ $]\!]$ =
    evaluate $e$
$\bar{\imath}$:  condjump $\bar{g}$
    frame $refs(S_2), vals(S_2)$
    execute $[\![S_2]\!]$
    pop
$\bar{j}$:  jump $\bar{h}$
$\bar{g}$:  frame $refs(S_1), vals(S_2)$
    execute $[\![S_1]\!]$
    pop
$\bar{h}$:  execute $[\![S_3]\!]$
```

Template 6 shows how the if-else-statement may be translated. The condition e is first evaluated. The if-statement without the else-clause is similar. The thing to notice about this template is that the order of which S_1 and S_2 appear has been reversed. This is because `condjump` only jumps if the condition evaluates to *true* and the else-clause must always be executed if that is not the case. The jump at label $\bar{j}$ ensures that S_1 is skipped. Finally, the next statement S_3 is evaluated.

template 7 Return Statement

```
execute$[\![$**return** $e$ $]\!]$ =
  evaluate $[\![e]\!]$
  returnV $refs(param), values(param)$
```

In template 7 the value of the expression is evaluated and the number of arguments to pop off the stack are given by the $refs, vals$ functions. This template omits the small detail that `returnR` is used in place of `returnV` if returning a reference type.

Expressions

template 8 Assignment Expression

```
execute$\llbracket ve \texttt{ = } e \rrbracket$ =
    evaluate $\llbracket ve \rrbracket$
    evaluate $\llbracket e \rrbracket$
    estore $\mathit{offset}(ve)$
```

Template 8 shows how an assignment is translated. The $\mathit{offset}(x)$ function provides the correct offset for the assignee.

template 9 Binary Operator Expression

```
execute$\llbracket e1 \text{ op } e2 \rrbracket$ =
    evaluate $\llbracket e1 \rrbracket$
    evaluate $\llbracket e2 \rrbracket$
    iadd
```

In template 9 the two expressions are first evaluated and the instruction for the operation is called, here `iadd` should be replaced with the correct instruction, if the binary expression isn't an integer addition.

template 10 Anonymous Function Expression

```
execute$\llbracket \textbf{(}\ param\ \textbf{)-}{>}\ id\ \textbf{:}\ S\ \rrbracket$ =
$\bar{j}$:  jump $\bar{n}$
$\bar{s}$:  frame $refs(S), vals(S)$
     argload $refs(param), vals(param)$
     execute $\llbracket S \rrbracket$
$\bar{n}$:  pushf $0$
     cfunc $\bar{s}$
```

Template 10 is similar to template 4. Yet, here the anonymous function is allocated and it is pushed onto the stack in the same template.

As mentioned earlier, these code templates are helpful when designing instructions. Laying out the basic foundation for the code templates in advance eases the implementation. Of course, the actual implementations tend to be longer and more complicated than the templates, but exhibit equivalent behaviour. In this case, our code generator implements the code templates without significant derivation. It does, however, require a visitor to be run before actual code generation, i.e. the allocation visitor must decorate the AST with information regarding vtable entries, which includes the number of reference type variables and value type variables, respectively.

Another useful feature is that code templates can be used to informally bridge the operational semantics to the implementation. We are basically defining how a given production of the abstract syntax is executed on the virtual machine. Though, these code templates are not defined well enough to be used as a formal method in their current state.

11.5.1 The Allocation Visitor

Before code generation can begin, groo must decorate the AST with information regarding the allocation of vTables, references, and values.

The allocation visitor makes use of the class `AllocationTable` to allocate `VariableDeclaration`s and manage scope conditions.

Allocation Table

The `AllocationTable` contains a scope, the root scope, and methods to enter and leave a scope, return the root frame allocation, and return the parameter allocation.

A scope holds the parameter allocation, frame allocation and the depth of the current scope. It also contains a next, which points to the scopes parent.

Whenever `allocate` is called, it increments either the current scope's frame's vTableEntries, referenceEntries, or primitiveEntries. Which one depends on the `VariableDeclaration` that is being allocated. If the `VariableDeclaration` is vTable allocated, the vTableEntries are incremented. If it is not vTable allocated, the allocation table checks if the type of the variable is a reference type or not, and increments the appropriate counter.

`enterScope` creates a new scope, setting the old scope as its parent. `leaveScope` on the other hand returns the current scope's `FrameAllocation`, which is intended to be set on the current `VariableDeclaration`. It also deletes the current scope, after updating the scope to be the parent scope.

Visiting The AST

The allocation visitor's objective is to visit all `VariableDeclaration`s. When the allocation visitor visits a `VariableDeclaration`, it will allocate the `VariableDeclaration`. Furthermore, the allocation visitor will also manage the scope structure, entering and exiting scopes as needed.

In all nodes containing parameters, the allocation visitor must record the `ParamAllocation` as well, which is similiar to the `FrameAllocation`, except it lacks the vTableEntries counter. A special case is the return statement, which also needs to know the `ParamAllocation` for its scope.

Part VI

Closing

CHAPTER 12

Discussion

In the following we will discuss the pros and cons of some of the decisions made during the development of the groo language.

Semantics

Looking back at the process of this project, we can conclude that we would have been well served by formalising groo earlier in the project. The formalisation of a proper semantics caused a number of changes in the grammar of the language - and consequently the abstract syntax tree. Needless to say, the semantics enabled us to define the language precisely, which got rid of a lot of uncertainty surrounding the implementation.

There is no distiction between fields and methods. This means that fields can act as functions if they are decared as one. It also means that both are implemented as closures, and subsequently each instance will have its own local version of the class methods, instead of pointing to a shared one. Obviously this is inefficient in terms of performance.

Syntax

Significant whitespace can be an issue in some cases, as the text editor used may insert a space, which could change the indentations. The parser will give feedback indicating where the syntax error is located, allowing the programmer to remove the extra space symbol. It could be necessary to fix this every time a program is written, to the annoyance of the programmer. It can also be argued that significant whitespace potentially lowers readability for large projects. In contrast for small code segments; forcing significant whitespace can improve readability for programmers not familiar with the code. It also forces programmers to keep methods small, maintaining the benefit of small code segments.

One could argue that the choice to prohibit multiple statements per line, yet allow multiple variable declarations per line, introduces inconsistency into the language. Sequential statements could be introduced, for example as $S_1; S_2$. Introduction of sequential statements would call for an "end of statement"-mark, for example the semicolon, as is used in many other languages. This would, however, introduce yet another form of inconsistency, as statements do not otherwise mark their conclusion.

Building Everything From Scratch vs. Using Tools

A lot of effort was put into constructing a lexer generator as well as a parser generator from scratch. Obviously, we could have used an existing and well-established tool to tokenize and parse program code. This would have saved time, as we would not have had to debug the generators and could have tested the grammar at an earlier stage.

On the other hand, it has given valuable experience and insight to lexing and parsing algorithms. It has also given us an understanding of writting grammar properly, both to avoid parsing conflicts, and to solve conflicts correctly. Had we not gained this understanding, we would likely have implemented various hacks to get the desired result. The lexer we implemented is also very fast, which may prove useful if we were to integrate it into an editor.

Language Priorities

In its current state, the groo language lacks object oriented features such as information hiding. This could have been prioritised higher, instead of putting so much work into enabling higher-order and first class functions. These two language features have also influenced the language a great deal, since we made a design choice to use closures for both function types and instance methods.

The grammar and type checker support enumeration. However, enumerations have not been implemented in the interpreter nor the virtual machine. Enumerations could have been sacrificed for other more important features, such as inheritance and arrays. Specifically, the virtual machine supports executing inheritance, however, the type checker has some issues with type inference. These issues could have been solved had the time spent on enumerations been allocated to inheritance instead.

Only a single conditional branch and a single loop statement were created to minimise the control structures in groo. More complex structures such as ″for″ statements, ″switch″ statements and ″do while″ statements were omitted to save time. These, more advanced control structures, can also be simulated with the current control structures.

Implementation Languages

We used Python to generate the C++ code for the lexer and parser. With Python it was relatively straightforward to implement the algorithms needed to generate a syntactical analyser. We did, however, have to optimise a lot of the code to get reasonably fast execution speeds for the generator. Nevertheless, performance was not crucial for the generator, as it is only run when the grammar or the tokens have been modified.

We implemented the compiler in C++. The C++ implementation is efficient and enables many low-level optimisations to be implemented in, for instance, the virtual machine. C++ also allows multiple inheritance which has simplified our AST. This language can, however, be difficult to write code in. Mishandling pointers can lead to very subtle errors, which are hard to debug. There is also no garbage collection, causing our implementation to have an odd memory leak here and there.

CHAPTER 13

Future Work

This section describes improvements that can be made to the groo language.

Sub-classing Inheritance

Sub-classing inheritance is possible in a groo program, but there is an issue with the type inference algorithm, which must be solved in order to fully support inheritance. Inheritance complicates type inference of variable declarations, since we have to solve the case where there may be multiple type candidates and we must find the minimal solution with the correct concrete type.

Both the operational and static semantics for groo will require some changes to enable inheritance. To begin with, this would require a modification of the abstract syntax to be able to specify a super class. Rules to recursively look for variables in the parent classes would need to be specified, as well as rules to ensure type safety with regard to inherited types.

Overloading

Method overloading allows us to have an arbitrary number of methods defined with the same name in a class, but with different arguments.

Operator overloading, however, can occasionally make code unreadable, e.g. if commonly used operators are overloaded and the meaning is not clear. It can make code more concise if it is not abused. For example, it could make sense to overload the arithmetic operators for objects representing vectors as this is a natural application of the concept.

There is currently a basic framework for overloading operators in the type checker. When type checking binary operators, a lookup will be performed to get type information for the operator. Type information for the standard operators is pre-loaded as part of the standard environment.

If overloading was to be implemented, the identification table would have to be modified, to allow for multiple occurrences of a method with the same name. An `OverloadedType` could be introduced, which would hold all the possible results of a `getAttribute` call. `isSubType` would then reduce the possibilities.

Type Coercion and Conversion

Type conversion is the act of changing the type of variables, also called typecasting. Type coercion is when this typecast is performed implicitly. Type casting and coercion may be beneficial when programming with numbers with varying precision. For example, when multiplying a floating-point number with an integer. There is the common case when there is the need to perform division with an integer as the denominator, which must be converted to a float to avoid obvious rounding errors due to loss of precision. Currently typecasting is not possible in groo and neither is coercion as it will result in a type error.

To allow the programmer to manually typecast, we would need to introduce a syntax for it. One approach could be the `c` language, where the explicit typecast is placed in parenthesis preceding the expression. An other way could be to use a method-like syntax, such as `(1+2).toType(float).` The latter approach is perhaps more explanatory, but may make expressions more complex than the first approach. Other languages, such as C#, also implement the `as` to perform an explicit typecast.

Type coercion would have to decide what type the immediate constituents of an expression must be converted to. We would have to introduce some kind of ordering on the primitive numerical types to be able to determine this. We could propose the following ordering: $int \leq float$. By default we would want the expression to evaluate to the type with the most precision. When we assign a value of greater type than the declared type of the assignee, for instance `x = 0.5f`, where `x` was previously declared to be of type `int`, we should be explicit, because we are performing a downcast in the assignment.

The type checker would have to make sure that a type cast is safe. Obviously, many type conversions should not be legal. For instance, converting an integer to float is legal, whereas from float to boolean is not. The typecast operation would also introduce some complexities when converting objects from one type to another. Introducing semantics to deal with this kind of operation may be very helpful.

Arrays, Lists and Tuples

Arrays, lists and tuples are groupings of entities. Working on a set of items is a very common task in any programming language, and arrays and lists are often used to do this. Tuples are groupings where the entities are not necessarily of the same type. This can be used to group unrelated items to return multiple values from a function. The benefit of this is that one is not required to define a new class to make this grouping possible.

Again, a syntax would need to be introduced for these data structures. Even though arrays and lists work very differently, a common syntax could be used for both to add consistency when working with them. Accessing an item within a list or array could have the form: `list[i]`. Moving to the next or previous item could look like `list[i++]` or `list[i--]`, respectively. Instantiating arrays and lists could have the syntax: `var list[4] = {1,2,3,4}` or `var list = {1,2,3,4}`, respectively, where the array can only hold four values.

String Manipulation

Easy string manipulation would be useful in a web programming language. For proper string manipulation we would need to support some kind of arrays. Strings could be an array of characters, which could be accessed in a fashion similar to arrays and lists. We could introduce a more specialised string syntax for strings and grouping data structures. Selecting sub-strings could have the form `string[:2-4:]`, which returns a substring containing the entire string except the fourth character (indexed by 3). Searching for a sub-string in another string could be done in the following fashion: `abracadabra["ra"]` that returns `(2,9)` as a tuple of the indexes where the sub-string occurs.

Null References

Currently object variables will always have to reference an object. It is very useful for a programmer to indicate that an object variable points to `null` or whatever name the null pointer is given in a particular language. For instance, if the programmer wants to access an object the variables referencing that object will have to point to something else otherwise unknown memory is going to be used. Changing the variable to NULL allows the programmer to test if the object the variable holds a reference to still exists.

Integrating HTML

Allowing HTML to be written in between code fragments could offer a scripting-like approach to creating web applications. The HTML mark-up should be ignored, perhaps by inserting escape keywords to indicate that the parser should ignore certain blocks. This would enable code to be embedded directly into the HTML mark-up and the programmer would not have to write all HTML programatically.

CHAPTER 14

Conclusion

We set out to create a language with a clear syntax and a set of minimalistic abstraction facilities. We implemented implicit typing of variables in order to aid readability and writeability of groo programs. Some of the abstraction facilities, such as objects, branching and control flow structures, higher-order- and first class functions, were implemented, whereas other important features need more work. For example, with proper support for strings, sub-classing and arrays, the language may offer a more mature and realistic application for web development.

We formalised the programming language and its type system with an abstract syntax, operational semantics, specifically big-step semantics, and static semantics allowed us to formalise the programming language and its type system. However, there is still work to be done in this area, as the type inference rules for implicitly typed variables require additional specification. Moreover, sub-classing is not described in the semantics, but it has been implemented in the language. Some object-oriented language features, such as information hiding, are still open issues in our language, both semantics and implementation-wise.

We were able to execute programs written in the groo language by means of recursive and iterative interpretation in the form of a virtual machine. The goal here was to ensure that the implementation defined program behaviour equivalent to the semantics of the language. The recursive interpreter makes use of abstractions that are very similar to the formal entities introduced in the semantics. This makes it relatively easy to follow the execution steps and verify that the interpreter is semantically equivalent.

With regard to the virtual machine, the code templates proved very useful for ensuring that the semantics were preserved during implementation. However, these code templates could have been strictly formalised using the abstract syntax. In addition, the virtual machine offered a more reasonable execution model, since it is more efficient and employs a garbage collection strategy.

We have shown that it is possible to create a statically typed object-oriented programming language from the ground up. This was accomplished by producing a grammar for the language and a lexer and parser which enabled us to translate program text into an abstract syntax tree. With this we were able to implement a type checker, which uncovers the type errors formalised by the static semantics for groo.

Part VII

Appendices

APPENDIX A

Code Samples

```
class MainClass:
    int _n = 7
    int main():
        facr(_n)
        fibr (_n)
        fac(_n)
        fib (_n)
        var anonFac = (int n)->int:
            return fac(n)
        var _fib  = fib
        anonFac(_n)
        _fib (_n)
        return 0
    int facr(int n):
        if (n==0):
            return 1
        return n * facr(n-1)
    int fac(int n):
        var f  = 1
        while (n>1):
            f  = f*n
            n = n-1
        return f
    int fibr (int n):
        if (n < 1):
            return 1
        return fibr (n-2) + fibr(n-1)
    int fib (int n):
        var f0  = 1,  f1  = 1,  f  = 1
        while(n > 0):
            f  = f0+f1
            f0  = f1
            f1  = f
            n = n-1
        return f1
```

Listing A.1: Groo implementations of the Factorial and Fibonacci functions. This example shows the use of recursion, anonymous functions and function pointers.

```
class MainClass:
    int main():
        html(”<html><head>\n<title>My Webpage</title>\n</head><body>\n”)
        html(”<h1>My homepage</h1>\n”)
        html(”<p>Welcome...</p>\n”)
        html(”<h3>My interests</h3>\n”)

        var data = ”<ul>\n”
        data = data + li (” Animals”)
        data = data + li (” Food”)
        data = data + li (” Programming”)
        data = data + ”</ul>”
        data = data + img(”images/hello.gif ”,” Hi there”)

        var div = tag(” div”)
        html(div(data))

        var i = 1
        var rows=””
        var table = tag(” table”)
        var options = Options()
        options.cssClass = ”odd”
        var oddrow = tagattr(” tr ”, options)
        var row = tag(” tr ”)
        var cell = tag(” td”)
        while(i <= 5):
            if (i %2==0):
                rows = rows + row(cell(” equal row”))
            else:
                rows = rows + oddrow(cell(”odd row”))
            i = i +1
        html(table(rows))

        html(a(” http:// intranet . cs.aau.dk”,” intranet . cs”))
        html(a(” mailto: foo@bar.com”,”email me”))

        html(”</body></html>”)
        return 0

    void html(string s):
        prints (s)

    (string,Options)−>((string)−>string) tagattr = (string el, Options opt)−>((string)−>string):
        var o = opt. attr ()
        var f = (string s)−>string:
            return ”<”+el+o+”>”+s+”</”+el+”>\n”
        return f

    (string)−>((string)−>string) tag = (string el)−>((string)−>string):
        var f = (string s)−>string:
            return ”<”+el+”>”+s+”</”+el+”>\n”
        return f

    (string,string)−>string img = (string src,string alt)−>string:
        return ”<img src=\’”+src+”\’ alt =\’”+alt +”\’ />\n”

    (string,string)−>string a = (string href,string s)−>string:
        return ”<a href=\’”+href+”\’>”+s+”</a>\n”

    string li (string s):
        var f = tag(” li ”)
        return f(s)

class Options:
    string cssClass
    string attr ():
        return ” class=\’”+cssClass+”\’ ”
```

Listing A.2: Groo implementation of a simple web page where various html elements are outputted.

Listing A.3: Output from code sample A.2

```
<html><head>
<title>My Webpage</title>
</head><body>
<h1>My homepage</h1>
<p>Welcome...</p>
<h3>My interests</h3>
<div><ul>
<li>Animals</li>
<li>Food</li>
<li>Programming</li>
</ul><img src='images/hello.gif' alt='Hi there' />
</div>
<table><tr class='odd' ><td>odd row</td>
</tr>
<tr><td>equal row</td>
</tr>
<tr class='odd' ><td>odd row</td>
</tr>
<tr><td>equal row</td>
</tr>
<tr class='odd' ><td>odd row</td>
</tr>
</table>
<a href='http://intranet.cs.aau.dk'>intranet.cs</a>
<a href='mailto:foo@bar.com'>email me</a>
</body></html>
```

```
class MainClass:
    int main():
        var dice = Dice()
        dice.initialize (NewRngFac)
        return dice.roll ()

class Dice:
    void initialize (void−>RngFac rngFactory):
        var fac = rngFactory()
        rng = fac.rng
    int−>int rng
    int seed = 5
    int roll ():
        seed = seed + 1
        return rng(seed)

class RngFac:
    int rng(int seed):
        return seed * 6 + 5

class NewRngFac extends RngFac:
    int prev = 0
    int rng(int seed):
        prev = seed
        return prev + (prev = seed) * 7
```

Listing A.4: Groo implementation of a random number generator. This example demonstrates inheritance (NewRngFac <: RngFac) a constructor (NewRngFac) can be passed as a function type. Note: this code does not generate random numbers in its current form.

APPENDIX B

Groo Intermediate Language Example

```
class MainClass:
        int main():
                return 40 + 2
```

Listing B.1: A simple groo program containing just MainClass.

```
0:  12
4:  jump      16
9:  vtable    4:
  12: 42
  13: 0
  14: 0
  15: 0
16: pushF     0
18: cmeth     0
21: call      0, 0
26: cmeth     0
29: call      0, 0
34: halt
35: vtable    4:
  38: 59
  39: 0
  40: 0
  41: 0
42: loadl     1:
  45:   38
  46:   0
  47:   0
  48:   0
49: new       0, 0
54: returnR   0, 0
59: frame     0, 0
64: argload   0, 0
69: loadl     1:
  72:   40
  73:   0
  74:   0
  75:   0
76: loadl     1:
  79:   2
  80:   0
  81:   0
  82:   0
83: iadd
84: returnV   0, 0
```

Listing B.2: gril code of B.1, 89 bytes.

Listing B.2 shows the gril representation of listing B.1. This is a simple example of a program that gives us the result of adding ’2’ to ’40’. n :, where n is an integer, denotes what is written in the nth byte of the gril code. Notice that following a `loadl` or `vtable` instruction there is some literal data; these are indented and prefixed with a byte number.

With this in mind, we can analyze the program. At byte 0 we can see the top level vtable, on byte 4 the first instruction is placed. From 4 we jump to 16, where the top level frame is pushed onto the reference stack. The next instruction is a `cmeth` instruction with offset, 0. This means that a closure for function is pushed with offset, 0, in the vtable of the topmost reference. The vtable for the topmost reference is 12, because we just pushed the toplevel frame onto the stack at position 16.

If the vtable is placed in 12 and we take offset 0 from that, results in reading the code store at the literal value of position in positions 12-15. This is 42, remember this, because the next instruction is a call instruction. It jumps to 42 where the literal 38 is loaded, then a new instruction in position 49 is executed, pushing a new object with 38 as vtable onto the stack. At line 54 this newly created object is returned. And we go back to where we called from i.e. position 21. Now we have just created an instance of the MainClass.

At position 26 a closure created from topmost object and code store position in its vtable with offset 0, is pushed onto the reference stack. The offset 0 in the vtable (at position 38) of the MainClass gives code store position 59. Calling a closure created with this, creates a new frame without variable allocations, this could be removed by optimisation, as could the following `argload`, where no arguments are loaded.

At byte 69 we load ’40’ onto the value stack, followed by the literal ’2’ on the value stack. Then `iadd`, adds the two integers and leaves the result on the value stack. Having done this, a value is returned from a parameterless function at byte 84.

This returns brings the execution to byte 34 where the execution halts, returning topmost value as the exit code. Thus, we have just ended execution with the result, ’42’.

Bibliography

Brad Abrams and Krzysztof Cwalina. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley Professional, 2nd edition edition, 2008. ISBN 978-0321545619.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Pearson, Addison Wesley, Boston, MA, 2 edition, 2006. ISBN 0-321-48681-1.

Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice & Experience*, 18(9), 1988.

Peter Bumbulis and Donald D. Cowan. Re2c - a more versatile scanner generator. *ACM Lett. Program. Lang. Syst*, 2:70–84, 1994.

Luca Cardelli. The computer science and engineering handbook. `http://lucacardelli.name/Papers/TypeSystems.pdf` acquired on 22/03/10., 2004.

Alan Demers and James Donahue. ”type-completeness” as a language principle. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 234–244, New York, NY, USA, 1980. ACM. ISBN 0-89791-011-7.

Hans Hüttel. *Pilen ved træets rod. Strukturel operationel semantik af programmeringssprog*. Books on Demand, 2010. ISBN 0006794262.

Jens Palsberg and Michael I. Schwartzbach. Three discussions on object-oriented typing. *SIGPLAN OOPS Mess.*, 3(2):31–38, 1992. ISSN 1055-6400.

Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994. ISBN 978-0471941286.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. ISBN 0262162091.

Lutz Prechelt. Are scripting languages any good? a validation of perl, python, rexx, and tcl against c, c++, and java. `http://page.mi.fu-berlin.de/prechelt/Biblio/jccpprt2_advances2003.pdf` acquired on 25/02/10., August 2002.

Robert W. Sebesta. *Concepts of Programming Languages*. Pearson Education, Boston, MA, 8 edition, 2008. ISBN 978-0-321-50968-0.

Michael Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, MA, 2 edition, 2006. ISBN 978-0-619-21764-8.

R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981. ISBN 0-13-709873-1.

David A Watt and Deryck F Brown. *Programming Languages processors IN JAVA*. Pearson Education, 2000. ISBN 13-978-0-13-025786-4.

Haiping Zhao. Hiphop for php: Move fast. `http://developers.facebook.com/news.php?story=358&blog=1` acquired on 25/02/10., February 2010.