Delft University of Technology

Grace in Spoofax

Michiel Haisma

TU Delft

Grace in Spoofax

by

Michiel Haisma

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday May 18, 2017 at 10:30 AM.

Student number:	1512285	
Project duration:	May 1, 2016 – May 18, 2017	
Thesis committee:	Prof. dr. E. Visser,	TU Delft, supervisor
	Dr. R. J. Krebbers,	TU Delft
	Dr. ir. G. Gousios,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

Abstract

Grace is a programming language that aims to be an example of a contemporary object-oriented language, to be used for teaching university level students. The language specification of Grace is informal, and its various implementations are difficult to comprehend and change. Spoofax Grace is an implementation of the Grace programming language, meant to serve both as a reference implementation, but also a specification, that can be easily read, understood and changed.

Spoofax Grace is implemented using the Spoofax language workbench, providing a declarative grammar, program transformations and dynamic semantics. From these specifications a language interpreter is generated that can execute Grace programs. The system covers the core aspects of Grace, yet a number of language features remain unimplemented. The implementation can be correlated to the informal Grace specification, and can be changed or extended at will.

Acknowledgements

I want to thank all members of the Programming Languages Research Group at TU Delft, in particular: Vlad Vergu, for creating and maintaining the DynSem language and his invaluable help creating the Grace dynamic semantics. Andrew Black, for helping define the Grace object model in DynSem; Timothy Jones, for his helpful pointers regarding specific Grace methods and local variables semantics. James Noble, Kim Bruce, Andrew Black, Micheal Homer *et al.* for creating the Grace programming language; Eelco Visser, for his supervision and support as my thesis supervisor. I would also like to thank the thesis committee: Eelco Visser, Robbert Krebbers and Georgios Gousios.

Part of this work was presented at the GRACE workshop at the 2016 ECOOP conference with the help of Vlad Vergu and Eelco Visser.[1]

Michiel Haisma
Delft, March 2017

[1]See: `http://2016.ecoop.org/event/grace-2016-spoofax-grace`

Contents

1
Introduction

The Grace programming language [3, 14] is a young programming language, used in educational environments for teaching object-oriented programming to university level students. For this language there exists an informal, prosaic language specification [4] and a number of implementations. These implementations are: Minigrace (compiler) [12], Kernan (interpreter) [32], Hopper (interpreter) [34]. The informal specification document describes syntax, program behaviour, and other language aspects. However, it is unclear to what degree the currently existing implementations conform to the informal specification. The reason for this is that the implementations are (1) not easy to read and understand, making it difficult to verify their conformance to the specification, and (2) are not defined in a declarative or formalised way such that certain properties may be shown to hold.

This work presents a language implementation for Grace called Spoofax Grace. Spoofax Grace aims to provide an implementation for Grace which is (1) readable enough such that it can be understood and compared to the language specification, and (2) is written using declarative tools. Spoofax Grace is both a language specification and an implementation, and it may be used for prototyping, testing, reviewing and verifying Grace language features and also for executing Grace programs.

The language implementation and specification is build with the Spoofax language workbench [38, 62], a collection of languages and tools that allows for effective language prototyping.

Spoofax Grace consists of the following parts:

- A Grammar created with SDF3 [31], a *meta-DSL* (Domain Specific Language) for creating context-free grammars. A parse table for an *SGLR* (Scannerless, Generalised, Left to right, Rightmost derivation) parser is automatically generated from this grammar.
- Program transformations written in Stratego [19], a meta-DSL for creating program transformations. These transformations simplify the Grace program *AST* (Abstract Syntax Tree) that resulted from parsing.
- Dynamic Semantics specified in DynSem [61]. This meta-DSL can be used to specify concise dynamic semantics. From this dynamic semantics specification an AST interpreter is automatically generated. This dynamic semantics specification is joint work with Vlad Vergu. Reverse engineering the Grace semantics and creating the initial revisions of the specification is work by the author. The specification in its current form is largely the work of Vlad Vergu. For more exact details on authorship of the DynSem specification, please review the Git commit history on `https://github.com/metaborgcube/metaborg-grace`
- A test suite to test syntax and program execution. Tests for testing the grammar and some transformations are written with help of SPT (SPoofax Testing language) [37]. This is a meta-DSL that allows for a concise specification of language tests. For testing the other transformations and dynamic semantics, tests are created on a file-by-file basis, and these tests are executed using

a parametrised JUnit [10] test runner. This file-based test suite for dynamic semantics testing is joint work with Vlad Vergu.

The test suite will be used to evaluate this implementation. In addition, a part of the system will be reviewed and compared to the informal language specification.

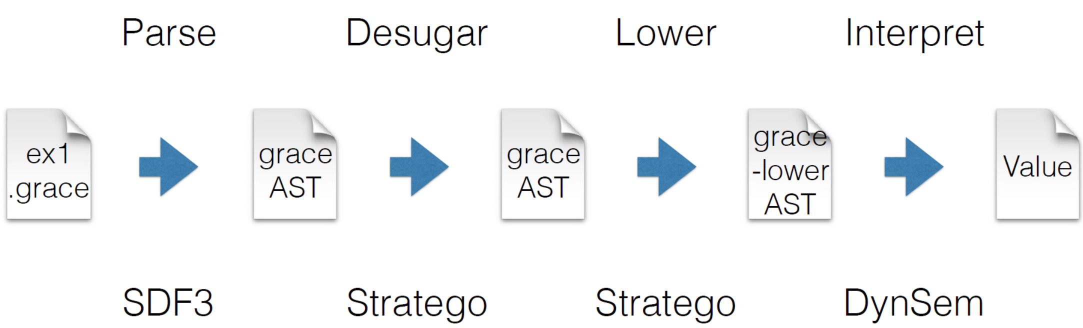
1.1. Architecture

A language implementation can be seen as a program (or a set of programs) that takes a program written in a certain language –in this case Grace– and the inputs, and evaluates the program, possibly generating some output. This includes any behavioural side-effects such as file IO or printing to `stdout`. To accomplish this, the implementation of the language must read, internalise and execute the program.

Spoofax Grace does this in four main steps: parsing, desugaring, lowering and evaluating. Each of theses steps is guided through a declarative specification: Parsing is done according to the SDF3 grammar, transformation is done through the specified Stratego rules, and execution is performed according to a DynSem specification. The programs that actually perform this are a proxy of these specifications: The SGLR parser uses a parse table generated from the SDF3 grammar. The Stratego rules are compiled to Java code, which is run after parsing. Finally the interpreter is generated from the DynSem specification. The tools provided by the Spoofax language engineering workbench allow these steps to be performed from a single environment. Generating the final interpreter from the DynSem specification combines the parser, transformations and the interpreter and allows Grace programs to be executed from single entry point.

The following diagram shows how these steps are implemented for Grace, these steps are explained in the following sections:

Figure 1.1: Process showing the steps when evaluating a Grace program



1.1.1. Parsing

The first step of evaluating a Grace program, is to parse the program code. This parsing step will yield a Grace AST that is used in subsequent steps. The grammar is specified in the SDF3 language, from which a parse table is generated. Spoofax includes an SGLR parser which uses this parse table to allow Grace programs to be parsed. Also, from the syntax definition, a number of AST signatures are generated. These signatures are used in the transformation and execution stages. If the given Grace program is syntactically valid and unambiguous, an AST is produced and passed on to the next stage: desugaring.

1.1.2. Desugaring and lowering

After parsing, we apply a number of transformation steps to the AST. The result will still be a valid Grace AST. These transformation rules are not very complicated, and are described as a number of Stratego [19] rules and traversal strategies that dictate how these transformation rules should be applied to the AST in what order.

Desugaring is explained in more detail in Section 4.2.

After the desugaring stage, the Grace AST is then lowered to the Grace-lowered language, a ‘core’ version of Grace that removes everything from the AST that is not strictly needed for evaluation and transforms nodes into into other nodes that more simple to use in the execution step. By using these new types of nodes in the AST, this Grace-lowered AST does no longer represent a valid Grace AST.

The lowering steps are explained in more detail in Section 4.3.

1.1.3. Execution

In this fourth step the Grace-lowered AST is interpreted by an AST interpreter, which is generated from the DynSem specification and is based on the Truffle [64, 65] framework and can be run on a normal JVM [45] or the Graal VM [66].

In Chapter 5 this step is explained in more detail and the dynamic semantics of the most important language features of Grace are highlighted.

1.1.4. Testing

The test suite developed with Spoofax Grace consists of two main parts: A set of tests written with the SPT framework. These tests are largely designed to test the syntax and some transformations. The other part of the test suite consists of a number of Grace programs with their expected outputs (or no output if an error is expected). Of these program tests, a number of tests come from the Minigrace [12] implementation. The other tests are written with the intention to exercise one single language feature or component at a time, making the test suite as granular as possible.

How testing is performed is explained in more detail in Section 6.1.

1.1.5. Evaluation

To evaluate this project, we consider how well Spoofax Grace performs at the test suite that is set up for this project. In addition, we consider how closely the Spoofax follows the Grace specification by evaluating an example throughout all the steps of the system. Finally, we very briefly look at how well the system performs in terms of language development and program execution.

The evaluation can be found in Chapter 6.

1.2. Outline

Chapter 2 shows the basic concepts of the Grace programming language are explained, to provide guidance when exploring the details of the Grace programming language.

Chapter 3 provides an overview of the syntax specification by explaining parts of the SDF3 grammar of Grace.

Chapter 4 explains how a parsed Grace program AST will be desugared (Section 4.2) and simplified further (lowered) into the Grace-lowered form (Section 4.3).

Chapter 5 explains the dynamic semantics of Grace and disuses how this was implemented using the DynSem language. This explanation includes object construction, inheritance, aliasing and excluding, method calls, non-local returns, declarations, confidentiality, dialects and imports, native operators, and types.

Chapter 6 shows the evaluation of this work, in particular we explain how the Grace specification compares to this implementation, by using a concrete example of Grace object construction and through discussing the test results. Since not all features of Grace are included in this implementation, the features omitted from Spoofax Grace are discussed in this chapter as well.

Chapter 7 discusses Grace’s other language implementations and other related work involving object oriented language implementations.

Chapter 8 discusses the conclusions drawn from this project, reflects on the process that has taken place and presents a number of possible future works. These include the possibilities for: implementing more syntactic features, setting up a generic Grace-testing suite, exploring the performance of this and other Grace implementations and how to make the Spoofax Grace implementation more publicly available for educational use.

The full specification of Grace is included in the appendices. These include the grammar (Appendix A), program transformations (Appendix B) and the dynamic semantics (Appendix C).

1.3. Code repository

All the source code for the Spoofax Grace implementation can be found in a git repository on: `https://github.com/MetaBorgCube/metaborg-grace/tree/045ac341d4`.

2

The Grace programming language

2.1. Origin of Grace

Grace has been designed by Andrew P. Black, Kim B. Bruce and James Noble, to be used for university entry- and intermediate level object-oriented programming classes. The designers of Grace explicitly name the different possible angle of attacks for these kinds of classes: object-oriented, functional or procedural, and all that with or without (static) types. Most importantly, it aims to be a good example of a contemporary object oriented language. To facilitate these multiple approaches, it can make use of dialects (a special form of importing) to change the available default methods, or enforce other (typing) constraints on a given program.

In addition to dialects, another appealing feature of Grace for use in education is its strict syntax, for instance: indentation in blocks is mandatory and line continuations must be even further indented. This forces students to write programs in a neat and consistent manner. In addition to these strict syntax rules, Grace still is a block-based language, meaning that every class, object, method, type, trait or block body is surrounded by curly braces (`{ }`) [14].

In contrast to some other object oriented programming languages such as Java, Grace programs can be written as *scripts*, meaning the top level of a Grace program can contain both declarations and statements, that will be executed upon program execution.

This chapter provides a simple overview of the Grace language, for more detail, please see the Grace language specification on: `http://web.cecs.pdx.edu/~grace/doc/`.

2.2. Objects

The main vehicle of a Grace program is the object. Objects in Grace are constructed using the object constructor. Unlike some object oriented languages such as Java, objects can be constructed without the need for a class. The idea for creating objects this particular way is borrowed from the Emerald language [13] and is also very prominent in a language like JavaSript [20].

Objects are first-class citizens, and therefore can be assigned to variables, passed as arguments, *etc.* Objects in Grace can be constructed and assigned to a constant as follows:

```
def cat = object {
    def colour = ”Black”
    var miceEaten := 0
    method eatMouse { miceEaten := miceEaten + 1 }
}
```

This object has two fields: `colour` and `miceEaten` and one method: `eatMouse`. The return value of a method is determined by the value of the last statement inside the method body, or the value that is returned by using the `return` statement, for more detail see Section 2.6.1.

All fields create an accessor method with the same name, and mutable fields (`var`) also get a writer method with the name '`<name>:=(_)`'. Fields can only be accessed or changed via these methods, however, they can be overridden by a subclass (see Section 2.7). The body of an object can contain arbitrary expressions that will be evaluated upon construction.

The structure of objects in Grace is immutable: the fields and methods (the structure) of objects in Grace cannot be changed dynamically after an object has been created. This is in contrast to many other object-oriented languages like JavaScript, Python or Ruby [20, 27, 59].

It is worth noting that in Grace, similar to JavaScript, O'Caml and Emerald, repeated execution of the same object constructor will yield *distinct* objects [13, 20, 44]. This is unlike some other languages that have a similar notation, like Scala and Self. [53, 58] Those languages will return the same object upon repeated evaluation of the same object constructor.

2.3. Classes

In Grace, classes can define what the structure of an object is. Class declarations look very similar to object constructors: they contain declarations and inline code of the object to be constructed. Essentially, class declarations are just syntactic sugar for factory methods that produce new objects, making the two following code snippets equivalent:

```
method Cat(name) {
    object {
        method meow { ”Meow!” }
    }
}

def myCat = Cat ”Franz”
```

Is equivalent to:

```
class Cat(name) {
    method meow { ”Meow!” }
}

def myCat = Cat ”Franz”
```

Creating an instance of the class `Cat` will just call the method `Cat`, that will return the last value of its body, being the constructed object. Also, a class definition can have parameters, which are directly available to the class' body, because it is actually just a parameter of the `Cat` method. Note that classes in Grace are not objects, nor are they types, classes are merely generators of objects. [14] This also means that objects created from a class, have no connection to the class or any other objects created from it.

2.4. Method requests

In Grace, nearly all computation is performed through method requests, or in the more traditional Smalltalk terminology this is also called "message sending" [28].

A method call is a request for an object to carry out one of its operations. This requires every method request to have some target object, also referred to as the *receiver*. In Grace, this receiver may either by explicit (the receiver is syntactically indicated), or implicit (the receiver is omitted from the actual request, but automatically resolved). The following example shows a method request with an explicit receiver:

```
a.f 5
```

In this example *a* is the also a method request, which will return an object, which is the receiver for the `f(_)` method request.

This means any identifier can be referred to as a method request:

```
f 5
```

This is called an *implicit* method call. Since all method calls in Grace must be resolved to a method declared on some object, the receiver object needs to be (dynamically) resolved.

This method identifier will be looked up in the following objects in order: the current object, current lexical scope and in lexically enclosing scopes, called *outer* scopes. For example:

```
method f(x) { x * 2 }
f 5 // will resolve to the f method
object {
    f 6 // will resolve to the same f method
}
```

Methods may be explicitly called on the object itself with the keyword `self`, or to a specific outer scope using the keyword `outer`. Requests that target an object more than one `outer` away may chain multiple outer calls together. Here is an example with an object that has a nested object inside:

```
1  method f { ”!” }
2  object {
3      method g { ”hello” }
4      object {
5          method h { ”world” }
6
7          self.h // ”world”
8          outer.g // ”hello”
9
10         f // ”!”
11         self.f // f cannot be resolved
12         outer.f // f cannot be resolved
13         outer.outer.f // ”!”
14     }
15 }
```

The method calls in the inner object perform the following operations, explained per relevant line number:

7 `self.h` requests method h on the current object.

8 `outer.g` requests method g on the object one scope outside of the current object.

10 `f` is an implicit request that resolves to method `f` on line 1.

11 `self.f` is an explicit request that calls the `f` method on the current object. Since this object has no method `f`, this is an *invalid* request. Because this request has an explicit receiver, there is no further lookup performed.

12 `outer.f` is an explicit request that calls to the `f` method on the outer object, but since this object has no `f` method (only g), this request is also *invalid*.

13 `outer.outer.f` is a chained outer request, that requests the `f` method on the second outerlying object. In this example that refers to the method `f` declared on line 1.

Note that once the receiver is made explicit, no further lookup is performed. Do note that often, the receiver of an explicit method request is an implicit request itself.

2.4.1. Field access

Calling methods and accessing fields are syntactically indistinguishable from each other, as can bee seen in the following example:

```
def cat = object {
    def colour is public = "black"
    method meow { print "meow" }
}

cat.colour
cat.meow
```

Here the field `colour`, and the method `meow` are defined as a constant field and a method on the `cat` object respectively , but their access pattern is identical. This is similar to the languages Eiffel and Self [49, 58]. This property of Grace allows programmers to abstract over how storage and computation is handled inside an object. In Eiffel, this principle is called: 'The Uniform Access Principle' as can be read in Meyer's *Touch of Class* [50]. The language Self also has this quality of having no distinction between accessing variables that are fields and method requests, although this language follows the prototype concept more strongly [58].

Note that when accessing constants and variables declared inside a method, and method arguments, these are not object fields but local variables. However, the patterns of accessing these local variables are the same as for normal method calls.

2.4.2. Confidentiality

There are essentially two modes of confidentiality in Grace that determine how methods on objects may be requested: *confidential* and *public*. By default, methods defined on an object are marked implicitly as `public`, and therefore these methods can be requested whenever there is access to the object. Methods may be marked with the `confidential` annotation to make them confidential.

Confidential methods can only be accessed by objects that have inherited from this object, or from an enclosed (nested) lexical scope. This is referred to as requesting a method from the *inside*. Requesting a confidential method from the *outside* (not from an enclosed lexical scope or to a method that is not inherited) will lead to an error.

Accessor and writer methods for variables and constants are marked as confidential by default. Optionally, constants may be marked `readable` (making its accessor public), and variables may be marked either `readable` or `writeable`, to make its accessor and/or writer method publicly available.

Note that unlike the `private` annotation in Java, objects that are created from the same class cannot access each other's confidential methods.

The following example illustrates how confidentiality can be used:

```
def b = object {
    def colour is public = "black" // marked as public
    var miceEaten := 0 // confidential by default
    method eatMouse { miceEaten := miceEaten + 1 } // public by default
    object {
        miceEaten := 2; // valid because access is from inside
    };
}

b.colour // valid because member is public
b.miceEaten // invalid because variables are confidential
b.eatMouse // valid because methods are publicly accessible
```

In this example, we declare an object with a constant, variable and a method. The constant `colour` is explicitly marked as public. The variable `miceEaten` is not annotated, which makes it confidential. The method `eatMouse` is not annotated as well, but it is public by default.

In the object `b`, we create a nested, anonymous, object and access the `confidential` `miceEaten` variable, this is valid because we are doing it from a scope which is nested in the scope of where the variable was declared (inside). When we access the `colour` member of `b`, we do it from the outside, since we are not lexically enclosed by the `b` object, nor have we inherited from it. Because `colour` is annotated `is public`, this access is valid. However, when we attempt to access the `miceEaten` member, the access is denied because variables are `confidential` by default, and we access from the outside. Finally, the invocation of the `eatMouse` method is valid because methods are public by default.

2.5. Blocks

In Grace you can define blocks, which are like lambda's. Blocks can have zero or more arguments, the following example shows how to create blocks:

```
def a = { print "hello" } // a block without arguments
def b = { x -> print ("hello, " ++ x) } // a block with a single argument

a.apply // evaluate block
b.apply "world" // evaluate block with argument
```

In this example we create two blocks, one without any arguments, and one with an argument. Requesting the `apply` method on a block will execute it. When applicable, the arguments are also passed with the `apply` method. Like method calls, the value of a block application is determined by the value of the last expression in the block. Blocks close over their lexical scope.

2.6. Control flow

The Grace language has no build-in syntax features for common control flow structures such as *if-then-else* or *while*. However, the `standardGrace` dialect provides a couple of methods that will allow for control flow, that can mimic the common control flow structures. The following method defines if-then-else in Grace using a build-in method `ifTrue(_)ifFalse(_)` on booleans, which conditionally applies one of the blocks passed as its arguments:

```
method if (cond) then (blk1) else (blk2) {
    cond.ifTrue { blk1.apply } ifFalse { blk2.apply }
}
```

The method declared in this example mixes its identifier names and arguments, this is called *mixfix* syntax. This allows for method requests that mimic control-flow structures as they are often used in many other languages. This example shows how the method from above may be requested:

```
if true then { // this is a block
    print "yay"
} else { // this is also a block
    print "boohoo"
}
```

Notice that this method request looks extremely similar to how such a control flow structure would look like in another programming language. The first argument of this method call is a boolean literal. The 2nd and 3rd argument to this method are blocks without arguments. The *implementation* of this method as seen above uses the build-in method `ifTrue(_)ifFalse(_)` on the boolean condition to apply the appropriate block.

2.6.1. Return

The `return` statement can be used to return immediately from a method or block. This statement has a special behaviour that allows control flow to break from block execution and return from the enclosing method. Consider the example from before, but now with a return statement added:

```
if true then {
    return false
    print "yay"
} else {
    print "boohoo"
}
```

The `return` statement will abruptly terminate the block application, but also will it return from the `if(_)then(_)else(_)` method. This allows for the creation of varied custom control flow structures.

2.7. Reuse

In Grace there are two main forms of reuse: object inheritance and trait usage.

Object inheritance allows an object to inherit from another freshly created object whereas trait usage allows objects to obtain methods from (possibly multiple) trait objects. The main difference is that traits may only contain methods and use other traits. This restriction is not in place for inherited objects, but objects may inherit at most from a single other object.

2.7.1. Inheritance

Grace's inheritance model is based on a principle called *uniform identity*. [35] This model is very similar to the behaviour of a class-based inheritance model, much like that of Java, but rather based on objects than on classes. For more details about dialects, see Section 2.9. Objects can inherit from other object, by requesting a method that returns a *fresh* object to a call in the `inherit` clause of the object constructor, this can be seen in the following example:

```
class A {
    method f { "hello" }
}
object {
    inherit A
    f // "hello"
}
```

Class A (which is just a method, see Section 2.3) will return an object when requested, and the bottom object is extended with the declarations from A. During object construction, the only object identity that exists, is that of the bottom object. Overriding is possible when an inheriting object implements a method with the same signature as a method declared in a parent object. This signature is only based on the name and arity of the object.

We expand the previous example to show how to override a method from an inherited object:

```
class A {
    method f { "hello" }
}
object {
    inherit A
    method f { "bye" }
    f // "bye"
}
```

In this example, both the parent class A and the to-be constructed object declare a method f. In this case, the bottom-most method overrides any methods coming from parent objects.

The inheritance system of Grace works as follows: Firstly, upon object construction, the bottommost object is created. Secondly, the inheritance clause is evaluated, this evaluation leads to the construction of the parent object, but no new identity is assigned to this object. When the topmost parent is reached, all fields and methods of the objects are created top-down. Finally, all initialisers and inline code is executed, again in a top-down fashion, and in the context of the bottommost object.

This design allows for down-calls from parent objects to overridden methods in subclasses. This particular way of dealing with object initialisation and inheritance is similar to that of Java [29].

2.7.2. Traits

Another form of reuse in Grace is the *trait* system. With the trait system, an object may obtain behaviour described in multiple, different traits. Traits are similar to objects, but they may only contain method declarations and use other traits. Traits can be constructed similarly to normal objects, as is shown in the following example where we construct two traits and use them in another object:

```
trait T1 {
    method square(x) { x*x }
}
trait T2 {
    method double(x) { x+x }
}

object {
    use T1
    use T2
    square 6 // 36
    double 6 // 12
}
```

The final object constructed here uses traits T1 and T2, and therefore has access to the methods described in the respective traits.

During object construction, trait usage is meant to be *symmetrical*. This means that the order of trait usage does not matter, and neither trait may override a declaration from another trait. Conflicts may be resolved by changing the names of the imported methods (aliasing), or excluding them. This particular style of multiple inheritance is called *Method Transformations* [35].

Similarly to inheritance, all declarations that come from traits are installed on the object before any initialisation or inline code is executed.

2.8. Type system

Grace has a structural type system. Each object's type is defined by the set of method signatures that it has. The method signature do not include the argument's types nor the return type. However, type object do contain the argument and return-type information, and types can be checked in a number of places: constant and variable assignment, parameter passing and returning from methods.

Types can be declared through a type object like follows:

```
type T = type {
    +++ (other: Number) -> Number
}
```

In this example we construct a type T specified by the `type { ... }` expression. This type expression contains a list of type signatures. The type signatures contain the method name, arguments, argument types (optionally) and return type (optionally) for a given method. Now this type can be referenced by its given name T.

Constants, variables and method parameters and returns can be annotated with types, as is shown in the following example:

```
def a: Foo = ...
var b: Bar := ...
method f (x: Number) -> Number { ... }
```

Type conformance is defined as follows: Given a type A and B. If type B conforms to type A, it means that B at least has all the signatures of A, and possibly more. Because the type system is strictly structural, there is no notion of inheritance or trait usages in type signatures.

Grace is meant to be gradually typed: Types may be optionally added, such that they can be statically checked, or they may be omitted in favour of dynamic type checking.

The Grace type system has more features like type arguments and type expressions. For more details and a full overview of all features in Grace please refer to the Grace language specification, which can be found at: `http://web.cecs.pdx.edu/~grace/doc/lang-spec/`

2.9. Imports and Dialects

Every Grace program, represented by some file containing Grace code is called a module. Every module is considered to be surrounded by an invisible `object { ... }` constructor. This means all top level declarations are actually members of what is called the *module object*.

At the top level, one can import another Grace module object by importing it and binding it to a name as follows:

```
import "animals/cats" as cats
```

This code will import the Grace module from a `"animals/cats.grace"` file, evaluate it, and bind it to a confidential constant field `cats`. Now cats is an object like any other object would be referenced.

Grace's dialect system is a special form of importing other Grace modules as part of the current program. A dialect may be imported as follows:

```
dialect "beginner"
```

This will load a file `"beginner.grace"`, evaluate it, and the module object from `"beginner.grace"` will now serve as the lexically enclosing scope of our program. Every top level declaration from `"beginner.grace"` will now be available to our program by implicit requests, since those declarations are reachable through the outer scope.

By defining library methods in dialects, the style of programming that is used in a Grace program can be varied by using different dialects. As well as providing declarations, the creators of Grace also intend to use the dialect system to create *pluggable checkers*; type checkers that can alter the behaviour of the type checker, by implementing specific *checker methods*. A checker method would be passed the AST of the program, and can perform any checks that it needs. [33]

In the next chapter we start explaining the Spoofax Grace specification, the first topic that is considered is the syntax of Grace.

3 Syntax

The Grace programming language has a very clean syntax with some very interesting features. The grammar of Spoofax Grace is created with SDF3. This meta-language allows us to specify a context-free grammar with disambiguation and preference rules. The disambiguation rules specify operator precedence to pick between different AST (Abstract Syntax Tree) branches, and preference rules to discard invalid ASTs after parsing. The Spoofax language engineering workbench generates a number of artefacts from this grammar: a parse table which is used by the SGLR (Scannerless Generalised Left-to-Right) parser, and a pretty-printer that can be used to transform ASTs back into concrete syntax (code formatter). Finally, it generates signatures for the constructors used in the AST that are used in the Stratego transformations and the DynSem specification.

In this chapter, we first highlight some interesting syntactic features of Grace, and subsequently the SDF3 syntax is highlighted to show how some of these features are declared [57].

3.1. Syntactic constructs

In the following subsections we highlight a number of interesting syntactic features of Grace.

3.1.1. Mixfix

One of the most obvious syntactic features of Grace is the use of *mixfix* syntax: Mixing method identifiers and parameters to form a multi-part method identifier. In Grace this is used to create methods that have syntactic structures similar to `if` and `while` in a language such as Java [29]. This allows method calls in Grace to look like this:

```
if (condition) then { foo(5) } else { bar(6) }
```

While this may look like this is a build in syntactic construct, it is actually a just an implicit request of the `if(_)then(_)else(_)` method. The second and third argument to this method calls are merely blocks, and no parentheses are needed in this case.

3.1.2. Operator methods

Another interesting feature of Grace is that it allows methods to be constructed using a number of operator characters[1], and those methods can subsequently be used with *infix* style method calls. For example:

```
method ++ (other) { self.append(other) }
```

[1] Any of the following characters: ! ? # % ˆ & | ˜ = + − * / \> < : . $

This method can be used as such:

```
bar ++ foo
```

This allows for very natural composition of expressions. An example of this can be seen in the Minigrace parser expression grammar: The ~ (tilde) operator is used to indicate subsequent valid constructs for a given expression [11].

3.1.3. Implicit method calls

Method calls in Grace can have many forms, but what makes it a really clean syntax is that any identifier can be a method call:

```
foo // just an identifier
foo 5 // identifier with argument
foo 5 bar 6 // mixed identifiers and arguments (mixfix)
```

Although all method calls in Grace need to have a receiver, these method calls have an implicit receiver. For more details how method call receivers are resolved, please see Section 5.4.

3.1.4. Layout sensitivity

Even though not supported by Spoofax Grace, the specification describes a number layout-sensitivity properties that together with implicit (mixfix) method calls, keep the syntax of Grace programs very neat and suitable for novice programmers. Most importantly, newlines determine when statements end. Consider the previous example, each of these three method calls are separated by newlines, which makes it clear that they are in fact three different method calls. In Spoofax Grace, layout is stripped during parsing, and therefore has no meaning. Fortunately, Grace allows for optional semicolons (`;`) to indicate the end of a statement. In Spoofax Grace, these are not optional, but mandatory instead.

In addition to ending statements with newlines, the Grace specification also prescribes that each block of code needs to be indented with more (but consistent) indentation than a previously opened block. This leads to neatly formatted code. In Spoofax Grace, these rules are not enforced.

3.1.5. Unicode characters

The specification describes a number of Unicode[2] characters that are used as part of the Grace grammar. These can be used instead of other more common ASCII character combinations. These include right-arrow (`->`), comparison operators (>=, <=, `!=`) and double brackets (`[[`, `]]`). Keywords are specified in the ASCII subset of Unicode, while other Unicode characters can be used for identifiers or operators. Due to technical limitations of Spoofax, this implementation does not support Unicode.

3.2. Syntax

In this section the implementation of a number of Grace syntactic features is explained.

3.2.1. Program

The goal of parsing is to transform a Grace program into an AST. The SDF3 (the meta-DSL that describes the grammar), grammar consist of a list of production rules. These rules are of the following form:

```
Sort.Constructor = Expression
```

[2]Unicode character set. See: `http://www.unicode.org/`

The expression can contain a template, which is a piece of concrete syntax delimited by <> or []. In these templates other production rules are referenced by escaping the template with the same delimiters.

The root node of the AST is the `Program` term, this term contains a sequence of statements:

```
Program.Program = <<{Statement "\n"}*>>
```

This template describes that a `Program` constructor must consist of 0 or more `Statement` productions, separated by newlines. Note that since layout is ignored during parsing, the newline separator here is only relevant for pretty-printing.

Statements can be any of following: dialect, import, declaration or an expression: In SDF3, we can write these rules as being of the same sort (`Statement`) but with a different constructor each time.

```
Statement.Dialect      = <dialect <STRING>;>
Statement.Import       = <import <STRING> as <Identifier><Annotations>;>
Statement.Declaration  = <<Declaration>;>
Statement.Expression   = <<Exp>;>
```

This forms the main structure of a Grace program. In the following subsections we highlight a couple of interesting syntactic constructs.

3.2.2. Object constructors

A very common expression in Grace is the object constructor, which has the following production rule:

```
Exp.ObjectDecl = <
                  object {
                      <Inherit><Use*><{Statement "\n"}*>
                  }
                 >
```

Within the template of this rule, we can see that the keyword (`object` and delimiters (`{ ... }`) of the object constructor are described in concrete syntax. Inside the delimiters there are three references (escaped with <>)to other sorts that go in these places: the inherit clause, zero or more use clauses, and zero or more statements. Formatting matters when forming production rules, this is picked up by the pretty-printer. One can see that in the body of the object, there must be an `Inherit` clause, zero or more `Use` productions, and zero or more Statement `productions`.

3.2.3. Method requests

In Grace, there are many forms of method requests. For brevity in the code, We also refer to method requests as method calls. The following grammar rules highlight explicit (with indicated receiver) and implicit (without indicated receiver) method requests respectively:

```
Exp.MCallWDot        = <<Exp>.<Part+>> {left}
Exp.MCallImpl        = <<Part+>>       {left}
```

Note that there is an annotation that implies that productions are left-associative (terms will be grouped from left to right in absence of parentheses). Because of the mixfix notation in Grace, method requests consist of one or more *parts*. As can be seen in the following piece of the grammar: Parts combine an identifier and arguments:

```
Part.Part = <<Identifier><CallArgs>>
```

`CallArgs` can be one of the following things: a literal (without parentheses), blocks, or one or more expressions within parentheses. The latter form is the more common one, as it is seen many programming languages. A literal can be a number, string, list or boolean. This list of available `CallArgs` options is expressed as follows:

```
CallArgs.ArgNumber = < <String>>
CallArgs.ArgBlock  = < <Block>>
CallArgs.ArgsParen = < (<{Exp ", "}+>)>
...
```

The first production rule shows that a string can follow directly, the second production rule indicates that a block can also be used as a direct arguement. The third production rule indicates that `CallArgs` can be a list of arbitrary expressions, separated by commas and surrounded by parentheses.

3.2.4. Binary operators

Arithmetic is done in a very generic way in Grace. Consider the following production rule for *infix* (the placement of an operator between two operands) expressions:

```
Exp.MCallOpEx = <<Exp> <Operator> <Exp>> {left}
```

In this production rule we can see that there is no concrete syntax specified, but just the placement of a reference to an `Operator` production between two references to `Exp` productions. `Operator` is defined ultimately by a regular expression in the lexical grammar, which can include the operators as specified in the Grace language specification [4]. For more details regarding lexical syntax, see Section 3.2.7.

3.2.5. Types

Named types can be constructed as follows:

```
Declaration.TypeDecl = <type <Ident><TypeArg> <Annos> = <TypeDeclBody>>
```

Here we see the keyword `type` followed by an identifier, type arguments and annotations, then the equals sign and the body of the type declaration.

The type declaration body can be either a type block, or a type expression:

```
TypeDeclBody.TypeDeclBlock = <<TypeBlock>>
TypeDeclBody.TypeDeclExp   = <<TypeExp>>
```

A type block is a list of `TypeRule` surrounded by curly braces.

```
TypeBlock.TypeBlock = <{<{TypeRule "\n"}*>}>
```

Finally, a type rule is a a method name optionally followed by a return type.

```
TypeRule.TypeRule       = <<MethodNames> <TypeRuleRightHand>;>
TypeRuleRightHand.RH    = [ -> [TypeExp]]
TypeRuleRightHand.NoRH  = <>
```

Note the use of different brackets to notate the type templates. For the RH constructor we use square brackets rather than angle brackets to prevent a conflict with the literal arrow (->) inside the rule.

3.2.6. Priorities

An important aspect of dealing with context-free grammars is that conflicts may arise when a piece of code is being parsed. In a language with many different types of expressions such as Grace, the following conflict would arise:

```
A.f + 24
```

Could be parsed as either of the following:

```
A.(f + 24)
```

Or:

```
(A.f) + 24
```

To decide between these kinds of ambiguities, we can indicate precedence of production rules in a separate section of the grammar. These encode needed priorities, but not real operator precedence (such as * takes precedence over +), since those are not directly encoded in the syntax definition.

```
context-free priorities
  ...
  Exp.MCallWDot >
  Exp.MCallPrefixOpExp >
  Exp.MCallOpEx >
  Exp.MCallOpExAssign >
  Exp.TypeExp
  ...
```

This snippet illustrates that explicit method calls are selected before method calls with a prefix operator, are selected before method calls with an operator, *etc.*

3.2.7. Lexical syntax

The main grammar of Spoofax Grace is concerned with the composition of different production rules. However, some sorts are defined using a different notation, very similar to programming with regular expressions. These rules form the lexical syntax. In Spoofax Grace, lexical syntax is used to indicate how identifiers, numbers, strings, comments, and layout is formed. Note that even though lexical syntax can be specified in a different way, the SGLR parser does not have a separate lexer.

For example, this is the rule that indicates the shape of identifiers:

```
lexical syntax

  ID = [a-zA-Z] [a-zA-Z0-9\'\_]* ":="?
```

As can be read from this: the identifier must begin with a letter (lower- or upper case), followed by zero or more letters, numbers, single quotes and underscores and may optionally end with '`:=`'.

For a full overview of the SDF3 grammar please see Appendix A.

After parsing is complete and a valid AST has been formed, the AST is now subject to a number of transformations, which are discussed in the next chapter.

4

Transformations

When a Grace program is parsed, the AST that is produced contains many different kinds of nodes (or constructors). This means that even though semantically these parts of the AST mean the same thing, they can have different forms. To illustrate this problem, consider the following three method requests:

```
print "hey";
print("hey");
print "hey" and "there";
```

After parsing, these method calls are represented by the following ASTs:

```
MCallImpl([Part(ID("print"), ArgString("hey"))])
MCallImpl([Part(ID("print"), ArgsParen([String("hey")]))])
MCallImpl([
    Part(ID("print"), ArgString("hey")),
    Part(ID("and"), ArgString("there"))
])
```

As can be seen in this example, the arguments of the implicit method call can be supplied in different forms (`ArgString`, `ArgParen`, *etc.*). Moreover, the method identifiers are distributed over different AST nodes.

To simply the AST and make it more homogeneus, we employ two sets of program transformations: desugaring and lowering.

In the *desugaring* phase, Grace ASTs are transformed and simplified, and after the transformation, the resulting AST is still a valid Grace AST. No new AST constructs are introduced and the AST can still be pretty-printed into valid concrete syntax.

In the *lowering* phase, the simplifications are even more rigorous, and new AST constructors are introduced to further remove AST noise.[1] After lowering, the resulting AST can no longer be pretty-printed into concrete Grace syntax, however it can be transformed into its own concrete syntax for inspection or debugging purposes.

The following section shortly describes how the transformations are applied using Stratego, and in subsequent sections the transformations are discussed in more detail.

4.1. Setup

The transformation rules discussed in this section are made with the meta-DSL *Stratego* [19]. In stratego, we declares a set of rules, these rules generally *match* on certain parts of the AST and

[1] Parts of the AST that have no semantic meaning are referred to as AST noise, or syntactic noise.

construct new AST nodes. For example a Stratego rule may be created as follows:

```
desugar-operator: Operator(n) -> ID(n)
```

This rule consists of three main parts: the name (`desugar-operator`), the matching constructor (henceforth called term, `Operator(n)`) and the created term (`ID(n)`). This particular rule transforms the term named `Operator` into a new term called `ID`. The sub-term `n` is moved over to this new term.

Rules of the same name will be combined when the transformations are compiled into Java code into a single *strategy*. Strategies may also be applied from within other rules by surrounding the strategy name with angle brackets (<>) as follows:

```
desugar-operator: Operator(n) -> ID(<remove-quotes> n)
```

In this example the sub-term `n` will have the `remove-quotes` strategy applied to it.

Generally, all transformation rules described in the Spoofax Grace specification are applied *exhaustively* on the AST. This means that every strategy will be attempted to be applied to every part of the AST, until none can be applied any more.

To accomplish this, all necessary strategies are combined in a separate strategy, and this strategy is applied to the AST using a library-offered strategy (innermost). For more details on the Stratego language please refer to the Stratego reference manual [41].

4.2. Desugaring

The desugaring of the Grace AST produced by the parser is mainly involved in making the AST more homogeneous. The following subsections cover the most important desugarings.

4.2.1. Class to method

One of the most interesting desugaring steps in the Grace language, is the desugaring of the `class` construct. The `class` constructor is syntactic sugar for a method that returns a fresh object. So the following Grace code:

```
class foo {
    method f { };
};
```

Is equivalent to, and thus will be desugared to:

```
method foo {
    object {
        method f { };
    };
};
```

The following Stratego rule will be applied to achieve this:

```
ClassDecl(MethodName(mIDs), annotations, type, inh, use, code) ->
  MethodDecl(mIDs, annotations, type, MethodBody([
    Expression(ObjectDecl(inh,use,code))
  ]))
```

Note in this snippet, that the method body that is begin constructed, begins with `Expression(ObjectDecl(...))`, the object constructor. Also the `inherit` and `use` parts of the class declaration is transferred to the object constructor, along with the body of the class.

4.2.2. Canonical method names

A critical part of desugaring is to convert mix-fix method names into a single, canonical method name. This canonical name then forms the unique and final representation of the method signature.

Consider the following Grace code:

```
greet ("John", "Hello") from ("Sam");
```

This method call has the following AST:

```
MCallImpl([
    Part(
        ID("greet"),
        ArgsParen([String("John"), String("Hello")])
    ), Part(
        ID("from"),
        ArgsParen([String("Sam")])
    )
])
```

But after desugaring it will be:

```
MCallImpl([
    Part(
        ID("greet(_,_)from(_)"),
        ArgsParen([String("John"), String("Hello"), String("Sam")])
    )
])
```

The call consists of a single part now. All arguments are also collected in that first, single part. The name of the method still resembles the original mixfix construction, but it squashed into a single identifier. Since the () tokens in the identifier are not valid in Grace, when this code is pretty-printed, the parentheses are removed. However, this still uniquely identifies the method name. The same canonical name generation is performed on all other declarations.

4.2.3. Generating string interpolation code

String interpolation is handled initially by the syntax specification, but we want to remove any string interpolation specific nodes from the AST and convert the string interpolation to string concatenation operations. This prevents the subsequent phases (lowering and execution) from having to handle the many AST constructors that are involved with string interpolation. In the transformation phase this is converted to Grace code that used the string concatenation operator (++) and thus removes any referenced to AST nodes that are specific to string interpolation. Consider the following Grace code:

```
"hello {name}, it's me: {sender}";
```

This is the AST after parsing:

```
InterpolatedString(IntPolStr(
    "hello {",
    [IntPol(MCallImpl([Part(ID("name"), NoArgs())]),
        "}, it's me: {"
    )],
    IntPolEnd(MCallImpl([Part(ID("sender"), NoArgs())]),
        "}"
    )
))
```

And this gets desugared to:

```
MCallWDot(
    MCallWDot(
        MCallWDot(
            String("hello "),
            [Part(
                ID("++(_)"),
                ArgsParen([MCallImpl([Part(ID("name"), NoArgs())])])
            )]
        ),
        [Part(ID("++(_)"), ArgsParen([String(", it's me: ")]))]
    ),
    [Part(
        ID("++(_)"),
        ArgsParen([MCallImpl([Part(ID("sender"), NoArgs())])])
    )]
)
```

Even though the AST has grown in size, there is no more mention of any string interpolation specific constructors.

This means that the lowering and execution need not be concerned with those constructors, simplifying those operations.

4.2.4. Annotations

In the desugaring stage, the default annotations are placed on declarations. Constants (`def`), imports (`import`) and variables (`var`) get a `confidential` annotation. Methods (`method`) and classes (`class`) get `public` annotations by default.

After all defaults are set, there will be a second transformation that considers the type of declarations and annotations, and optimises them. To illustrate: declaring a variable will at run-time generate a getter and setter method, thus, a variable declaration with the `public` annotation will get with the `readable` and `writeable` annotations that will be applied to its getter and setter respectively. Similarly this is true for constants and imports, but only a getter is generated for these declarations and thus only the getter is affected.

4.2.5. Other steps

Besides the desugaring steps detailed above, other steps include:

- Trait declarations are desugared to methods that return a trait.
- Places where types that are not explicitly annotated will receive the dynamic type annotation.
- Match cases are desugared into if-then-else calls with type matches.
- Calls with operators (using the operator symbols instead of normal identifiers) are transformed into regular method calls.
- Prefix method names are transformed into normal method names.
- Blocks are desugared into a single form.
- Method arguments (single, no arguments, arguments in brackets, literals) are desugared into a construct that is equal for all calls.
- Double quote symbols (") are removed from string literals.

All these desugaring are applied exhaustively on the AST. The complete transformations can be found in Appendix B.

4.3. Lowering

After desugaring, the AST of a given Grace program has become more simple, but it still uses only valid Grace AST constructs. These are not yet free of AST noise, and can be put into simpler forms that make the dynamic semantics specification more concise. This is because the these transformations prevent the dynamic semantics from containing different rules for the same thing. These transformations effectively translate the Grace AST into a more simple language, hence it is referred to as *lowering*. The resulting Grace AST is said to be in the Grace-lowered 'language'.

This lowered language can be seen as a 'core' version of Grace, although technically it combines AST constructors from both Grace and an additional Grace-lowered grammar. The Grace-lowered AST should no longer contain certain constructors. Also, after lowering there is no longer a way to convert the AST to concrete Grace syntax. However, there still is a concrete syntax for Grace-lowered that can be pretty-printed, this can be used to translate for inspection and debugging purposes.

The following sections highlight the most important lowering steps that are performed.

4.3.1. Generalising

In the Grace syntax, there exist specialised AST nodes for some constructs. A good example of this is, that a method call with a literal as a single argument does not need parentheses. For all literals there is a separate constructor, but after lowering, all these different forms will be removed and the arguments are put in a list. This transformation leads to loss of information as to how the request was originally made.

The rules that perform these transformations are as follows:

```
lower-arguments: ArgNumber(a) -> [Number(a)]
lower-arguments: ArgString(a) -> [String(a)]
lower-arguments: ArgsParen(as) -> <lower-arguments> as
lower-arguments: [ArgsParen(a) | b] -> [a | <lower-arguments> b]
...
```

The `lower-arguments` rule is invoked from multiple rules where arguments present themselves in the AST, such as for blocks.

4.3.2. Simplifying

To simplify the AST, new AST constructors are introduced for many syntactic constructs. These are created in such a way that there is only one type of constructor needed for each type of construct. The following constructs have a specific, lowered version:

- Implicit and explicit method calls.
- Object constructors.
- Inherit and use expressions, including aliasing and exclusion.
- Blocks.
- Uninitialised expressions.
- Unknown types.
- Method, variable and constant declarations.
- Type rules.

The resulting AST is very explicit, as every possible sub-term for each AST node is present. For example, consider the following Grace program:

```
class A {
    method f(a) {
      print "Hey, " ++ a
    }
}

A.f "Jude"
```

This program can be desugared, lowered and pretty-printed into the following Grace lowered concrete syntax:

```
_method A |||| is public () : () -> _Unkwn {
    _object {
        _method f_ |||| is public (a) : (_Unkwn) -> _Unkwn {
            _recv (_impl (print_("Hey, "))).++_(_impl (a()));
        };
    };
};
_recv (_impl (A())).f_("Jude");
```

Note that in this example, the lists of type arguments (||||) are present, but empty. All declarations are annotated with a confidentiality modifier. All implicit types are explicated to the unknown (dynamic) type. All method calls have an identical form, and are either explicit or implicit (indicated by `_recv` or `_impl` respectively). Methods do not only have an argument list, but also a list of the same length with the types corresponding to those arguments.

Note that all lowering transformations happen after the desugaring operations. So in the example above, before any lowering rules were applied, first the fragment is desugared.

The full grammar for Grace-lowered can be found in Appendix A. The lowering transformations can be found in Appendix B.

Stratego is a very suitable language for composing these transformations, because it allows us to use generic tree traversals, and specify concise transformation rules.

However, it could be possible to move these transformations to the domain of the dynamic semantics. By doing this one would lose the possibility of using Stratego (and libraries) to perform static analysis tasks, but on the other hand it would require one less meta-DSL to be understood and used, even if it is not very complex.

After desugaring and lowering the AST is ready for execution. This next phase is discussed by explaining the dynamic semantics of Grace in the next chapter.

5

Dynamic semantics

The dynamic semantics of Spoofax Grace specifies the behaviour of Grace programs at run time. These specification are written in the Meta-DSL called *DynSem*. In Sections 5.1 to 5.10 of this chapter we dive deeper into the semantics of Grace, explaining the most crucial aspects of the language. In Section 5.11 of this chapter the basics of DynSem are explained. The DynSem specification is joint work with Vlad Vergu. As the the most recent DynSem implementation of these semantics are mostly implemented by Vlad Vergu[1] , the focus lies on the underlying semantics themselves, and in lesser detail how these are implemented. The inference rules shown in this chapter are based on the DynSem rules.

5.1. Program start-up

A Grace program is represented by a `.grace` file. This program, also called a *module* can be be executed. The contents of the file are treated as if they were to be inside of an `object` constructor.

The body of the program —which is a list of statements and declarations— will be evaluated and reduced to a value, which will be the final result of this Grace program. This can be seen in the following (simplified) rule:

$$\frac{ProgPath, R, O, S, P, Src \vdash p :: H, L, VH, DCache, ICache, EX \Rightarrow v :: H, L, VH, EX}{p@Program(\_) \overset{init}{\Rightarrow} (v, EX, H)} \tag{5.1}$$

Note that in addition to the *environment* variables passed through rules, as indicated by the symbols before the turnstile (⊢) symbol, there are also a number of components passed after the double colon (::). In a more traditional notation these would be propagated as a tuple.

Also note that this rule has an arrow named $init$. This indicates that this is the first rule that should be applied in the semantics. All other rules in the specification can be considered an *unordered set*, as is common in natural semantics [36].

Finally, note that in this rule, we *match* on the $Program(\_)$ AST node (also referred to as constructors), and we bind it to the p variable (indicated by the @-symbol).

The following list describes all the components used in this rule:

$ProgPath$ File path of the Grace module being executed.

R Return marker that indicated which method to return to upon return.

O Reference to the `outer` object. Initially points to a non-existing object.

[1]At the time of writing, around 21% of the lines of code in the dynamic semantics specification were last touched by the author. Please see the GitHub repository for more details: `https://github.com/MetaBorgCube/metaborg-grace/tree/045ac341d`

- S Reference to the `self` object. Initially points to a non-existing object.
- P Phase of execution; can be in normal execution mode or object construction mode. Initially in normal execution mode.
- Src Used to determine the source of methods (inherited, used, or from within object). Initially 0.
- H Heap (map from reference to value). Initially empty.
- L Locals (map from name to reference). Initially empty.
- VH Value heap, (map from reference to value, for locals). Initially empty.
- $DCache$ Dialect cache, since imported dialects may only be evaluated once, this cache stores the module object from used dialects in case other imported module re-use the same dialect. Initially empty.
- $ICache$ Import cache; similarly to dialects, imported module objects may only be evaluated once, so when multiple imports import the same module, the object is drawn from this cache rather than evaluated again.
- EX Exception, indication of early return, exception or normal status. Initially empty.

All these components will be initialised after this first rule. After this initial rule has been applied, the next (and only) rule that can be applied is the following:

$$\frac{collect\text{-}dialect\text{-}statement(prog) \Rightarrow dia \qquad load\text{-}dialect(dia) \Rightarrow S' \qquad S', O\, S \vdash code \Rightarrow v}{S \vdash prog@Program(code) \Rightarrow v} \tag{5.2}$$

In the conclusion, like in the previous example, the $prog$ variable is bound to the constructor that follows after the @-sign. In addition, the sub-term $code$ of that constructor is also immediately bound.

In the first premise, we call the meta-functions called $collect\text{-}dialect\text{-}statement$. Meta-functions are evaluated similarly as if they were any other constructor: they are matched against a rule, evaluated and bound to their result variable. This rule collects the optional dialect statement from the code body and binds it to the variable dia.

The second premise $load\text{-}dialect$ evaluates this dialect statement and results in a new `self` reference that is used in the next premise.

In the third premise, there are two components that come before the turnstile: S' and S. The latter is actually assigned to the component sort (type) O, and S is the variable name that we actually reference. What happens here is that the dialect object serves now as an outer scope for the module object being executed. Thus, this is also the premise that will lead to the evaluation the body of the program.

Note that in this rule, there is no mention of any other components that are not touched, this is the case for all rules in this chapter.

5.2. Code execution

In the previous section we showed that the evaluation of the $Program$ constructor will lead to the evaluation of its body. This body is a list, and thus we define a number of rules that operate on this list:

$$\frac{}{[\,] \Rightarrow DoneV()} \qquad \frac{c \Rightarrow v}{[c] \Rightarrow v} \qquad \frac{c \Rightarrow \_ \qquad cs \Rightarrow v}{[c|cs@[\_|\_]] \Rightarrow v} \tag{5.3}$$

These three rules are applicable in case of an empty body: (1) a body with a single entry, of which the value will be the value of this whole body, (2) a body with more than one entry or (3) an empty body, of which the result will be the DoneV value.

The following statements can occur in a code body:

Figure 5.1: Grace-lowered grammar rules for statements (simplified).

⟨*Statement*⟩ ::= object { ⟨*Inherit*⟩ ⟨*Use**⟩ ⟨*Statement**⟩ }
| impl ⟨*Identifier*⟩(⟨*Exp**⟩)
| recv ⟨*Exp*⟩.⟨*Identifier*⟩(⟨*Exp**⟩)
| method ⟨*Identifier*⟩ ⟨*TypeArg**⟩ ⟨*Annos*⟩ ⟨*Param**⟩ ⟨*ParamType**⟩ → ⟨*TypeExp*⟩ { ⟨*Statement**⟩ }
| def ⟨*Identifier*⟩ ⟨*TypeExp*⟩ ⟨*Annos*⟩
| var ⟨*Identifier*⟩ ⟨*TypeExp*⟩ ⟨*Annos*⟩
| block { ⟨*Param**⟩ ⟨*TypeExp**⟩ → ⟨*Statement**⟩ }

These statements are explained as follows:

object Object constructor. Has an inherit clause, use clauses and body statements.

impl Implicit method request. Has a name and argument expressions.

recv Method request with receiver. Has as receiver (an expression), name and argument expressions.

method Method declaration. Has a name, type arguments, annotations, parameters, parameter types, return type and body statements.

def Constant declaration. Has a name, type and a list of annotations.

var Variable declaration. Has a name, type and a list of annotations (Initialisation is a separate method request).

block Block constructor. Has a list of parameters with types and a list of body statements.

The following sections describe how these statements are handled in more detail. Note that throughout this chapter, the terms method *request* and method *call* are used interchangeably.

5.3. Object construction

When an object constructor node is encountered, it requires the interpreter to construct a model of this object. To illustrate the many steps that go into the object construction, please consider the following Grace code snippet, with numbers added in comments to each line. These numbers represent the order in which they are handled. Each step of the object construction is explained below.

```
class A {          //  3
    method f { } //  4
    f              //  5, 10
}
trait B {          //  7
    method g { } //  8
}

object {           //  1
    inherit A      //  2
    use B          //  6
    method f { } //  9
    g              // 11
}                  // 12
```

In this example we create a class, a trait and finally the object to be constructed. The class and trait are considered to be already handled before the object is constructed.

As can be seen in Figure 5.1, the object constructor is represented by the object constructor node, the inherit clause (optionally empty), a list of trait usage clauses (can be empty), and finally the body of the object (which itself is a list of statements).

The following list explains what happens during object construction in order:

1. The object constructor is considered, a new object is allocated.
2. The `inherit` expression is evaluated.
3. The method of the class declaration is resolved, but no new object is created.
4. Method `f` is allocated.
5. Initialisation code of this class is stored, but not executed.
6. The `use` expression is evaluated.
7. Method of the trait declaration is resolved, but no new object is created.
8. The method `g` is allocated.
9. The method `f` of the main object is allocated, overriding the previous declaration of `f`.
10. All allocations are complete, so the initialisation code is run, starting at the top object of the inheritance chain, so `f` is called, which resolves to the overridden method.
11. The `g` method is evaluated as part of the bottom object initialisation code.
12. Object construction is completed.

Important to notice is that the execution of the `inherit` clause must happen under a different mode than normally. This is where the P flag first mentioned in Equation (5.2) is for: before evaluating the expression in the inherit clause, we change the standard flag for one that indicates that we are in a special object construction mode. This ensures that we are going to construct the object and initialise it in the correct order. Since we have different flags that indicate these execution modes, we also have different execution rules.

The following two rules describe the behaviour of object initialisation for object construction in the $Exec$ (normal execution) mode and in the $Flatten$ (mid-object construction) mode respectively:

$$\frac{\begin{array}{cc} \textit{new-object}(S) \Rightarrow S' & \textit{snapshot-locals}() \Rightarrow L \\ S\,S', 0\,S, \textit{Flatten} \vdash \textit{inherit} \Rightarrow \textit{oc-inherit} & S\,S', 0\,S, \textit{Flatten} \vdash \textit{uses} \Rightarrow \textit{ocs-use} \\ \multicolumn{2}{c}{(S, L, \textit{oc-inherit}, \textit{ocs-use}, \textit{code}) \Rightarrow \textit{oc}} \\ S' \vdash \textit{install-members}(\textit{oc}) \Rightarrow \textit{oc}' & S' \vdash \textit{init-object}(\textit{oc}') \Rightarrow \_ \end{array}}{S, \textit{Exec} \vdash \textit{Object}(\textit{inherit}, \textit{uses}, \textit{code}) \Rightarrow S'} \tag{5.4}$$

In this rule, a new object is allocated, its reference is bound to S'. Locals are captured and bound to L. In $Flatten$ mode, the `inherit` and `use` clauses are evaluated and return an object constructor tuple and a list thereof respectively. An object constructor tuple for this object is created and bound to oc. Members will be installed, the object constructor tuple is used for this, this tuple also contains all necessary information to install members for all parents in correct order, this is abstracted by the *install-members* meta-function. Finally, the initialisation code is run by the *init-object* function, this information is also kept in the object constructor tuple resulting from the *install-members* function. The resulting value of an object constructor is an object reference, which is a value with an address that points to this object on the heap.

$$\frac{\begin{array}{cc} \multicolumn{2}{c}{\textit{snapshot-locals}() \Rightarrow L} \\ 0S, \textit{Flatten} \vdash \textit{inherit} \Rightarrow \textit{oc-inherit} & 0S, \textit{Flatten} \vdash \textit{uses} \Rightarrow \textit{ocs-use} \end{array}}{S, \textit{Flatten} \vdash \textit{Object}(\textit{inherit}, \textit{uses}, \textit{code}) \Rightarrow (S, L, \textit{oc-inherit}, \textit{ocs-use}, \textit{code})} \tag{5.5}$$

This rule captures the locals, and constructs the object constructor tuples for inheritance and use clauses. Finally it creates an object constructor tuple for the object itself.

5.3.1. Aliasing and exclusion

On the `inherit` and `use` expressions of the object constructor, there can appear any number of `alias` and `exclude` clauses. These clauses indicate the copying and removal of a method signature during the inheritance sequence, respectively. The general idea is to set extra components to the execution environment when installing methods on objects (in the rules above denoted by the *install-members* function). Note that in these procedures, the resulting methods must be checked for presence (in case of exclusion), and for duplication (in case of aliasing). When an excluded method is not present or a method is aliased to a name that is already defined, the execution is halted.

5.4. Method requests

The main way of evaluating expressions in Grace is through method requests (or calls). Method requests come in two forms after the lowering process: explicit (also referred to as qualified) and implicit. Explicit requests have an expression that evaluates to the specific receiver object that the method needs to be requested on, whereas implicit requests need a way to look up what method to invoke and on what object, since its receiver is not yet determined.

5.4.1. Qualified requests

Requests with an explicit receiver can be expressed as follows:

```
foo.f 12
```

In this example, `foo` is the receiver of this call, `f(_)` is the canonical method name, and `12` is the argument to `f`.

To evaluate this request, firstly the receiver must be resolved. In this case, the receiver expression is itself a method request, an implicit one with the name `foo`. This should return an object, which has a method with the name `f`. This will be requested directly after the arguments are evaluated from left to right.

The rule for evaluating qualified calls is as follows:

$$\frac{e \Rightarrow recv \qquad recv \neq buildin \qquad \textit{call-qualified}\,(recv, name, es) \Rightarrow v}{Exec \vdash MCallRecv\,(e, name, es) \Rightarrow v} \tag{5.6}$$

The second premise of this rule implies that the receiver is not one of the build-in object types (Number, Boolean *etc.*), in that case the call is handled by a separate function depending on which object. These functions typically delegate to native operations. The function *call-qualified* can be evaluated by on of the following two rules, one in case the function is applying a block directly and one in the generic case, where the method needs to be retrieved from the receiver object. Methods are stored on the heap as closures, these closures collect all necessary information (references, types, code body) to execute the methods it represents.

$$\frac{clos = closure \qquad x.startsWith\,(''apply'') \qquad call\,(clos, vs, name) \Rightarrow v}{\textit{call-qualified}\,(clos, name, vs) \Rightarrow v} \quad \frac{\textit{lookup-local-method}\,(recv, name) \Rightarrow clos \qquad disambiguate\,(clos, name) \Rightarrow \_ \qquad call\,(clos, vs, name) \Rightarrow v}{\textit{call-qualified}\,(clos, name, vs) \Rightarrow v} \tag{5.7}$$

The main reason for having these two rules split up, is that the `apply` method on blocks is never explicitly defined in a program. This means that when we would do a normal method lookup on the `apply` method, it would not be found. Instead we use two possible rule matches to check for this case. If any of the premises of either rule fails, the other rule will be applied.

In this rule a number of meta-functions are applied: The first one (*lookup-local-method*) is responsible for fetching the receiver object from the heap, and retrieving the correct method from it. The second premise uses the *disambiguate* meta-function, that will raise the appropriate error and halt the interpreter if the method returned in *clos* is not valid.

The last function to be discussed is the *call* meta-function. This rule actually performs the request, given the receiver, method name and argument values.

$$\frac{\begin{array}{c} clos = Closure\,(S, O, params, locals, code, R, paramtypes) \\ type\text{-}check\,(paramtypes, vs) \Rightarrow true \qquad ensure\text{-}access\,(name, clos, S) \Rightarrow true \\ add\text{-}locals\,(locals) \Rightarrow \_ \qquad update\text{-}locals\,(params, vs) \Rightarrow \_ \\ S, O \vdash handle\text{-}return\,(code, R) \Rightarrow v \end{array}}{call\,(clos, vs, name) :: L \Rightarrow v :: L} \tag{5.8}$$

This rule does the following: it checks the types of the arguments, ensures that the caller is allowed to access this method from an encapsulation perspective, adds the local variables from the declared scope, adds the arguments to the local variables, and finally, *handle-return* evaluates the code and checks whether to return normally or to propagate a return *exception* . This return exception will then be handled at the appropriate level.

Note that the `self` and `outer` references that come from the closure are set prior to executing code. This is to make ensure correct lexical scoping of the method body in question. In addition, note that this function preserves the local variables component L, and re-outputs it, this is to prevent any changes leaking into the callers scope.

5.4.2. Implicit requests

Implicitly calling methods without a receiver, requires the receiver object to be resolved before the method may be evaluated. The following Grace code shows an implicit method call:

```
foo
```

There can be multiple possible receivers to this method request, and the receiver is resolved in the following order:

1. Local: In case the name `foo` resolves to an argument of a method, or a constant or variable that is declared directly inside a method, `foo` will resolve to a local value. When a method is local, there is no receiver object. Locals are stored in an environment and have a reference to their value on a heap.
2. Current object: If the method `foo` is a field on the same object that the request is made from, the call resolves to that.
3. Outer object: The next possibility is that the method `foo` is declared in some outer scope, so all outer scopes up to the dialect scope need to be checked one by one for having the `foo` method.
4. Inherited or from trait: Since the object construction folds all inherited and used objects into a single object, methods from traits and the inherited object will be resolved to a field on the object itself.

This means we need to locate the method in three possible places: local scope, current object or some outer object. This is reflected in the rules for implicit calls:

$$\frac{\begin{array}{c} is\text{-}local\,(name) = true \\ access\text{-}local\,(name, vs) \Rightarrow v \end{array}}{call\text{-}implicit\,(name, vs) \Rightarrow v} \qquad \frac{\begin{array}{c} is\text{-}local\,(name) = false \\ lookup\text{-}local\text{-}method\,(name) \Rightarrow local \\ lookup\text{-}outer\text{-}method\,(name) \Rightarrow outer \\ disambiguate\,(local, outer, name) \Rightarrow clos \\ call\,(clos, vs, name) \Rightarrow v \end{array}}{call\text{-}implicit\,(name, vs) \Rightarrow v} \tag{5.9}$$

The first rule only matches if *name* is indeed a local. In this case we access that local and evaluate the request to a value.

The second rule shows what needs to happen in the case that *name* is not a local. In this rule we perform lookups in the current object, much like in Equation (5.7) but also in the outer scopes. Then there is a need to disambiguate between these two results: If either returned a valid closure, that is correct and that respective closure is evaluated. If neither lookup returned a closure, the method could not be found and the evaluation will halt. If both lookups return a closure, there might be a conflict: the Grace language does not allow the shadowing of methods from an outer scope with methods that come from an inherited (or used trait) object. This is also checked within the *disambiguate* function. Finally, the same *call* function is called and evaluated to a value as shown in Equation (5.8).

5.5. Returning

The `return` statements inside of blocks can have a special effect. When a method is applied from within a method, the `return` will not only return from the block, but also from the method. The following example highlights why that is useful:

```
method foo {
    if condition { // this is a block!
        return
    }
    print "hello?" // will not be executed
}
```

This example shows a method, that executes a request to the `if(_)` method, and passes in a block as an argument. The block however has a return statement in it. When the return statement is executed, it not only returns from the block, but also from the method that requested its execution. This behaviour prevents the `print` method from being called, and allows the programmer not having to guard all subsequent statements from begin executed when a block returns.

5.6. Declarations

In Grace, we can define constants (`def`), variables (`var`), methods (`method`) and imports (`import`). Of these declarations, constants and variables can occur both within an object and a method body. This context is important to the way they are handled: within objects they will be stored as fields, whereas declarations in a method context will be stored as a value in an environment. Note that methods can only be declared within objects, and imports may only be defined at the top level of a module (which is also an object). These two possible different contexts are explained in the following sections.

5.6.1. Object context

When a method declaration is encountered within the context of an object, the method is stored as a field on the object as a closure. The object will be stored on the heap. When a method is requested, the appropriate closure is retrieved and can be evaluated. These closures store the following information:

- Reference to the object the method is declared in.
- Reference to the outer object the method is declared in.
- Parameters as a list of identifiers.
- Body of the method.
- The local environment.
- Whether the method is confidential or not.
- Whether this method is created as an inherited, used or normal method.
- Where to return to if a return were to occur.

Constant declarations are essentially a method declaration, where the constant defines a getter method and a slot on the object which holds a reference to the value that is the result of evaluating the initialisation expression of the constant.

This is the rule for defining a constant on an object:

$$\frac{\begin{array}{cc} \mathit{add\text{-}slot}(\mathit{name}) \Rightarrow i & \mathit{has\text{-}readable}(\mathit{annos}) \Rightarrow \mathit{public} \\ \mathit{make\text{-}getter}(\mathit{name}, i, \mathit{public}) \Rightarrow \mathit{getter} & \mathit{install\text{-}method}(\mathit{getter}) \Rightarrow \_ \end{array}}{\mathit{install\text{-}declaration}(\mathit{Constant}(\mathit{name}, \mathit{type}, \mathit{annos}, e)) \Rightarrow \mathit{SlotWrite}(i, e, \mathit{type})} \tag{5.10}$$

Note that since this regards *object* context, this rule is part of the object construction phase, as discussed earlier in Section 5.3. This rule will be invoked as part of the $install\text{-}members$ rule as seen in Equation (5.4).

Firstly, this rule creates a new numbered slot on the current object and binds it to the variable i. Secondly, it checks whether the getter for this constant should be declared public using the $has\text{-}readable$ function. The result (which is either true or false) gets bound to $public$. Thirdly, the meta-function $make\text{-}getter$ creates a getter method and binds it to $getter$. Finally, the getter method is installed on the object.

What is very interesting about this rule (and similar rules, like the one for variables) is that what this rule returns is a new constructor. This constructor is saved for later evaluation. This is important because the object initialisation semantics of Grace prescribe that all declarations need to be handled before any initialisation or inline code is run. Evaluating the expression of the constant declaration is of course part of this initialisation, thus it must not happen at this time.

Variable declarations are similar to constants, but for variables also a setter method is installed. Setting the initial value (if given) is separated from the declaration in the lowering phase.

Imports are very similar to constants but instead of evaluating the expression, a meta-function is used to read an (module) object from an external file, or get it from a cache if that file has already been imported.

5.6.2. Method context

When declarations are evaluated within a method scope, the values are not stored as fields on objects, but they are kept in an environment. This environment (or component) is passed through such that subsequent statements and lower lying scopes have access to locals from surrounding (outer) scopes.

The rule for declaring constants inside the context of a method is shown below:

$$\frac{e \Rightarrow v \qquad type\text{-}check(type, v) = true \qquad update\text{-}local(name, v) \Rightarrow \_}{Constant(name, type, \_, e)) \Rightarrow v} \tag{5.11}$$

This rule evaluates the expression of the constant immediately, checks it against the declared type and stores this local using the $update\text{-}local$ function. Any annotations that are declared on this constant will be ignored (hence the underscore in the rule conclusion): even if this declaration is said to be `confidential`, that is irrelevant because it cannot be accessed from the outside regardless.

Note that Grace does not allow methods to be declared inside methods, and because imports may only occur at the module object, only constants and variables can be declared as locals.

5.7. Confidentiality

In Grace, declarations (constants, variables, methods, imports) can have have their access limited, as is discussed in Section 2.4.2. In this section we discuss how this confidentiality is applied and enforced.

5.7.1. Annotations

Declarations may be annotated with `public`, `confidential`, `readable` and `writeable`. These annotations control from what objects methods can be called. As mentioned in Section 5.6.2 these annotations only affect object fields, making them relevant only for method calls, not for local variable access.

Essentially, there are only two concepts of confidentiality in Grace: public and confidential.

Confidential methods declared on an object means that a method can be accessed from either: inside the same object, inside an object inside the declaring object (a nested object), or from an inheriting object (an object that inherits from another object or uses a trait that declared the method).

Public methods can be accessed by any object that has access to the object the method is defined on.

For local variables, there is no extra encapsulation needed, even though you can annotate `var` and `def`, it does not effect the semantics, as you are never able to reach a local variable from outside the scope they are defined in.

When declaring a method, the desugaring step ensures that any method has either a `public` or a `confidential` annotation. This information is transferred into the method closure, also see Section 5.6.1. For constants, the annotation can be either `readable` or `confidential`, which will result in the getter closure being public or confidential respectively. For variables, the possibilities are either: `confidential`, `readable`, or `writeable`. This means the getter method will be confidential, public or public respectively and the setter closure will be confidential, confidential or public respectively.

5.7.2. Checking confidentiality

When evaluating a method call, the following invariant holds: a resolved method defined on an object is either public or confidential. When declarations are handled, this boolean is set to the proper value (see also Section 5.6). To check this at run-time we consider this public boolean value and check whether the call is being performed from the *inside* (see also Section 2.4.2). When an access violation is detected, (a method is confidential but accessed from the outside) the interpreter halts with an error.

5.8. Dialects and imports

There are two main ways of referencing code from other files in Grace: dialects and imports. Dialects can even alter the way the language is perceived, because the dialect typically provides access to library methods. A Grace module has one dialect, if one is not specified, it uses the dialect `standardGrace`. Since in Grace even simple control flow is handled through methods declared in a dialect, this heavily influences the way a program is written. Imports are handled in a more familiar fashion: other Grace files can be imported by referencing their filename and are assigned to an identifier. A Grace module can have zero or more imports. This sections discusses how these two methods of importing are implemented.

5.8.1. Dialects

As seen in Section 5.3, importing a dialect needs to be handled before the body of a Grace module is executed, because it affects the surrounding scope. This can be done by checking the code at the top of the Grace module for the presence of the `Dialect` constructor. If this constructor is not present, the default dialect `standardGrace` is loaded instead.

When loading the dialect, it needs to be checked whether that dialect has not been loaded before (this can happen through imports), as the Grace specification specifies that any import is only evaluated once. To achieve this, there is a semantic component that maps from dialect names to object references. When the dialect is not present in this cache, it is loaded from the file system, parsed, transformed and evaluated. After evaluation, a reference to the dialect object is added to the cache.

The reading, parsing and transforming of the dialect is performed through a native operator.

To prevent the leaking of the dialect's dialect into deeper lying modules, the connection between the outer scope of object is cut by updating that reference with a reference to an empty object.

When the `none` dialect is requested an empty object is returned as a dialect. For instance, the `standardGrace` dialect uses the none dialect.

5.8.2. Imports

Imports work in a similar way as dialects, but they are defined through a statement that can be executed normally. This is very similar to how a constant is defined, because imports are named. After the external module is evaluated it is bound to its given name. Like dialects, when evaluating an import statement, a cache is used to prevent duplicate evaluation of the same module, which is not allowed by the Grace specification.

5.9. Native operators

To implement native operators we use specific values in the specification for: numbers, strings, booleans *etc.*

When a receiver of a method call is found to be one of these, the method call does not proceed to look up the appropriate method, but checks a number of build-in method handlers. These handlers subsequently use a number of native operations defined in Java classes to perform operations like number arithmetic and boolean logic.

In addition to handling native functions for a couple of data-types, there are a number of build-in implicit methods. These can be used for handling certain implicit method calls that need to defer to a native operation (like `print`), but also to optimise functions that would otherwise might suffer from a stack overflow problem. An example of a method like this is the `while(_)do(_)` method. This method can easily be created as part of the standard dialect in a recursive form, but it would not be effective for doing many iterations, as this will cause a stack overflow. By implementing loops as build-in methods, we can leverage tail recursion elimination to prevent the stack from growing too large and increasing performance.

5.9.1. Limitations

With the current approach to dealing with native operations, it is not possible to override the default behaviour of these methods. When the receiver of the object is resolved to one of the aforementioned data-types, the method call is intercepted and normal lookup semantics no longer apply.

A different approach to native data types can be taken to support extension. This can be done through implementing the native types as proper Grace objects in the standard dialect, and having methods that themselves call into native operations.

5.10. Types

Types in Grace are structural, they are expressed as a set of method signatures as follows:

```
type A = {
    f(x: Number) -> String
}
```

In addition to specifying types as list of signatures, one can compose types with type expressions and use a type constructor to create ad-hoc types:

```
def b: A | type { g -> Boolean } = ...
```

In this example we annotate the type of the constant b with a type expression: <type> | <type> (type variant expression). This means the value of b must conform to either the type defined by A or the ad-hoc constructed type: `type { g -> Boolean }`.

To be able to dynamically check these kind of types, the type annotations of declarations are stored as the type expression itself. Resolution of a type name is identical to resolving a method call, except that now a type value is expected to be the result of the look-up.

The type expression is compared with the dynamically computed type. When the type conforms, the program proceeds, when it does not, the interpreter is halted.

Type declarations such as the one above, declare a type object. The name of the type is stored in the same namespace as any other declaration (Grace only has a single namespace).

5.11. DynSem

All dynamic semantics for Spoofax Grace are specified in the meta-DSL DynSem [61]. The rules presented in this chapter are a (simplified) representation of inference rules as specified in the Spoofax

Grace DynSem specification.

A DynSem specification consists of the following parts: *signatures*, *sorts*, *arrows*, *rules* and *components*. In this section, DynSem will be explained briefly. For further explanation of the DynSem language please refer to the official documentation [60].

Signatures These are the constructors that are defined in the syntax of Grace and Grace-lowered. A constructor consists of a name and optionally a number of subterms. More constructors can be added at will.

Sorts The type of a constructor is referred to as its sort. For example: the constructor `MCall` is of sort `Exp`. These sorts are defined by the syntax and can be added as well.

Arrows Arrows define the reductions that may occur in the specification. The arrows specify a source and target *sort*. Optionally, an arrow may be named.

Rules Rules form the implementation of the reductions the arrows define. Rules make up the majority of the specification. They match an arrow by specifying a conclusion that conforms to the from- and to sort and the arrow name. In addition to a conclusion, the rule can have zero or more *premises*. These premises are be evaluated as well, and can form restrictions on whether a certain rule can be applied or not. These rules are roughly similar to big-step style operational semantics [21], but the premises follow the conclusion rather than the other way around. To indicate this, the premises are preceded by the `where` keyword.

Components As shown in the rules in this chapter, rules are evaluated accompanied by components, also referred to as semantic components. In DynSem there are two forms of components: read-only (appears before the turnstile in a rule) and read-write (appears after the double colon in a rule). In a rule, the read-write component will be propagated from each premise to the next up until the result of the conclusion, whereas the read-only component will only be applied to the premises, but not between them, and not through the conclusion.

The following example shows a very simple DynSem specification:

```
module example

signature

  sorts
    Program
    Value
    Expression

  arrows
    Program --> Value // Unnamed arrow
    Expression -e-> Boolean // Boolean is a build-in sort

  constructors
    Prog: Expression -> Value // Prog(Expression) is of sort Value
    Result: Value // Result() is of sort Value

rules

  Prog(exp) --> Result() // Conclusion of rule
  where
    exp -e-> true. // Rules end with a full stop

...
```

In this example specification we define three sorts, two arrows (one without name and one named e), two constructors and a single rule. This rule conforms to the unnamed arrow, as its conclusion describes a reduction from the Program sort to the Value sort. In this rule, the subterm of Prog will be bound to the name exp. In the premise of this rule, exp should be reduced to true. Finally, as the product of the rule, the Result() constructor is produced.

Multiple rules can apply to the same arrow. The generated interpreter will match terms with rules, and will try to fulfil the premises. If a premise fails, it will backtrack and attempt to a apply another rule. If no rule can be applied, the interpreter is halted.

5.11.1. Implicit reductions

A key feature of DynSem that enables concise and readable specifications are implicit reductions: these are reductions that can take place without an explicit premise that indicates that a certain term has to be reduced.

The following example changes the rule above into one that has an implicit coercion:

```
rules

  Prog(true) --> Result().
```

Because there exists a possible reduction from the sort Expression to Boolean using the -e-> arrow, and the given subterm of Prog should be Exp but a Boolean is given, DynSem will implicitly add this reduction as a premise. This is indicated in the Spoofax IDE as a note that informs the implementer of what implicit reduction is being applied here. This is similar to how it is shown in the example above with a blue, squiggly, underline.

5.11.2. Components

In the following example we have a rule with a read-only component (A) and a read-write component (H). Note that these names represent be the sort of the component as well as the variable its bound to.

```
rules

  A |- Program(subterm) :: H --> v
  where
    A {} |- some-function(H) --> v :: H';
    A != H'.
```

When evaluating the first premise, we *set* the read-only component of sort A to be the empty set, and from the result of this evaluation we *get* a new read-write component H'. The next premise forces a check that A is not equal to H' can either succeed or fail. Components that are not mentioned explicitly may still be passed implicitly. This is one of the great aspects of DynSem, as these implicit propagations make sure each rule only needs to mention the components it needs to access or affect. This prevents rules from having their true meaning obscured by superfluous details and specify only what is relevant for that rule.

5.11.3. Abrupt termination

Another addition in DynSem that allows rules to stay concise and readable is the automatic handling of abrupt termination. In the case a return statement is encountered, subsequent statements must be prevented from being executed. This can be achieved by setting a semantic component to have a certain value, and requiring all other rules to not match on this component. The rule in Spoofax Grace for this is as follows:

```
components

  EX : Exn = Ok() // component EX of is of sort Exn, default value is Ok()
```

DynSem allows us to declare a special component that has a *default* value. Now the interpreter will check after each subsequent evaluation of a premise inside a rule, that this component is still at its default value. Additional rules may be created to handle the exceptional cases. This allows us to specify rules with multiple premises, without having to manually check every time the exception component may have changed.

5.12. Process

Initially, the first implementation that was created for Spoofax Grace already followed the pattern of parsing, desugaring, lowering and evaluation. To get to a state in which the most simple Grace programs could be run, each of those four components were minimal, but functional. The syntax did not include types, to simplify the grammar and reduce ambiguity problems. The transformations were very simple, and since many components of the AST were not used in the evaluation at that point (such as annotations, exclusions, aliases, and imports) these were ignored in the DynSem specification. The DynSem wildcard identifier (_) proved very useful here.

Also this initial development iteration was very monolithic, most of the semantics were contained in a single specification file. This was acceptable at this point because the specification was only a few hundred lines of DynSem code, but it would not be wise to continue down this path.

Finally, there was no automated way of testing the dynamic semantics specification, so it was not possible to detect all regressions when changing the specification.

In the second iteration, these problems were attacked: The grammar was extended to include types, almost all syntactic constructs got a form of simplification and the dynamic semantics were split up in many more modular parts. Also, a large number of tests (see: Chapter 6) was added to be able to monitor the specification for regressions.

During the project, DynSem gained support for: printing reduction rule stack traces, improved notation and handling of semantic components and support for abrupt termination with default values for semantic components. During the project, the DynSem version was regularly updated, and this required the specification be updated as well whenever a breaking change was introduced.

The next chapter details the evaluation of this project by discussing the test suite and how well the Spoofax Grace correlates with the Grace language specification.

6

Evaluation

The goal set out for this project is to create an implementation and specification for the Grace programming language. The system should conform to the Grace specification, have usable performance and it should be easy to understand and change. In this chapter we consider the test suite used to evaluate Spoofax Grace (Section 6.1). In addition, we consider how Spoofax Grace correlates to the the Grace language specification and how it complies to another implementation's test set (Section 6.3). Also, we discuss what features are not included in the Spoofax Grace implementation (Section 6.4). Finally, we have a brief look at the performance of this system, not only how Grace programs perform but also the time Spoofax Grace and other implementations take to rebuild after a change has been made (Section 6.5).

6.1. Testing

The language implementation of Spoofax Grace includes parsing, transformation and execution components. A correct implementation of these components does not only rely on the grammar, transformation specifications and the dynamic semantics specification, but also on the generated parser, the compiled transformations and the generated interpreter. To verify that each component is behaving correctly, we run a large number of tests against the specification. These tests allow the specification to be changed without letting regressions go unnoticed. These tests allows the language implementer to be more confident that the specification is implemented correctly and no unforeseen bugs arise.

6.1.1. Syntax and transformation testing with SPT

To maintain confidence that the language specification is not regressing while being implemented, a test suite consisting of 427 small tests is used to test the grammar and transformations. Spoofax provides a way to supply language tests, using the SPT (SPoofax Testing language). This language let us set up snippets of concrete Grace code with the respective expected behaviour. These tests are collected in SPT files, and can be grouped as the language implementer sees fit. An SPT file may look like this:

```
module syntax-numbers

language grace

start symbol Program

test decimal number with leading dot [[var a := .5;]] 0 warnings 0 errors
...
```

This test has a name, a concrete code snippet and an expected result. If there is any problem with parsing or analysing this code snippet, the expected result would not hold and this information is reported to the language implementer.

Other test expectations can include: indication of a parse failure, to be equal to the parse result of another code snippet or to be equal to a given AST.

The tests are divided in the following categories:

- Desugaring - Assignments
- Desugaring - Canonical names
- Syntax - Types - Declarations
- Syntax - Types - Expressions
- Syntax - Types - Methods
- Syntax - Types - Variables and constants
- Syntax - Ambiguous
- Syntax - Assignments
- Syntax - Blocks
- Syntax - Classes
- Syntax - Comments
- Syntax - Expressions
- Syntax - Identifiers
- Syntax - Match case
- Syntax - Methods
- Syntax - Numbers
- Syntax - Objects
- Syntax - Precedence
- Syntax - Program
- Syntax - Traits
- Syntax - Types
- Syntax - Visibility

In total there are 427 SPT tests spread across 22 files. Running all these tests takes about 10 seconds, or about 0.04 seconds per test.[1]

6.1.2. Program evaluation with JUnit

In addition to the SPT tests, there are a large number of unit tests that are created to test the dynamic semantics and program transformations that are not covered by the SPT tests. There is a parametric JUnit test that will scan a specific folder for `.grace` files and `.output` files with the corresponding names. These files are grouped together in a test-case. The JUnit test-runner will then run all these tests as normal Grace programs, and compare the program output with the expected output in the `.output` file. The absence of an equally-named `.output` file indicates that this test-case should not exit normally, but with an exception that halts the interpreter. The reason for having a separate infrastructure for dynamic semantics testing is twofold: initially, there was no support in SPT to test language execution. Additionally, it makes the test set more portable, such that other language implementations (that do not use Spoofax) can also benefit from these tests.

Each of these tests is made to test only one aspect of the Grace language, as far as that is possible. Although for many test cases it depends on other language features to be working as expected. For instance: to test inheritance it requires at least method declarations, method calls and object construction to be working. Almost all tests require some form of program output using the `print` method. This means even though the test are as small as possible, many form integral tests. The program tests are joint work with Vlad Vergu.[2]

In addition to the test cases that were created for this project specifically, there are a number of tests taken from the Minigrace implementation. These tests were written in the GUnit (Grace-unit) testing framework, which relies on many advanced Grace features to be working. These advanced features include: exception handling, imports and correct scoping. Moreover, these tests were grouped in big Grace source files, with up to 50 tests in single file. This is problematic because if there is a problem with one of the tests , none of the test cases in the entire file could be run. This prevents the language to be worked on incrementally, improving and implementing the language over time. To alleviate these issues, and be able to run all tests before these features were completed, the Minigrace tests were extracted and put into the form as described above. This way these tests could be used with fewer

[1] These results were achieved with Windows 10 x64, on an Intel i7-6700k (4.2GHz) with 16GB of memory using Oracle JRE 1.8.0-92.

[2] In the program test suite, 47% of the lines were last modified by the author. Fore more details, see the repository: `https://github.com/MetaBorgCube/metaborg-grace/tree/045ac341d4/grace.interpreter/src/test/resources`

dependencies on other Grace features. Also, the test result provides and indication of completion of the language implementation.

In total there are 390 semantic tests specified. Of all semantic tests, 134 are from the Minigrace tests. The remaining 254 tests were created to gradually test the Spoofax Grace implementation. They are split up in the following categories:

- Object model - 48 tests
- Scoping - 31 tests
- Methods - 18 tests
- Expressions - 27 tests
- Qualification - 13 tests
- Exceptions - 3 tests
- Visibility - 22 tests
- Traits - 12 tests
- Aliasing - 24 tests
- Control flow - 6 tests
- Imports - 10 tests
- Types - 38 tests
- Whole programs - 2 tests

These tests can also be used to test any other Grace implementation. An initial investigation shows that there exist compatibility differences between the implementations (and between the implementations and current informal specification). For instance: one implementation (Hopper) only supports classes in the `A.B` format (defines a constant (`A`) that is an object with a method (`B`) that returns fresh objects. Currently, this format is no longer part of the Grace language (Spoofax grace supports it for backwards compatibility). This shows it might be difficult to compare tests across multiple implementations. This is also discussed in the discussion (Section 8.1.3).

6.2. Review of Specification

Evaluating the readability and quality of the specification is not trivial because it is not easy to quantify. We attempt to show the specification is readable by examining a critical part of the Grace language (the object constructor) and considering each part of the specification and comparing it to the Grace language specification. As with all constructs in the Spoofax Grace specification, object construction consists of four main parts: Syntax, desugaring, lowering and dynamic semantics.

Syntax When we consider the concrete syntax of a Grace object constructor in Source code 6.1, and the syntax definition in SDF3 in Source code 6.2 it becomes clear that the definition resembles the concrete syntax very closely.

Source code 6.1: Grace object constructor.

```
object {
    inherit A;
    use B;
    method f { };
};
```

Source code 6.2: Definition of object constructor in SDF3.

```
Exp.ObjectDecl =
<
 object {
     <Inherit><Use*><{Statement "\n"}*>
 }
>
```

This is a very common pattern in the SDF3 Grace grammar. The full syntax definition can be found in Appendix A.

Transformations The object constructor requires no desugaring, only lowering. The following Stratego rule defines the lowering of the object constructor from an `Object` to an `ObjectL`:

```
lower-objectdecl:
  ObjectDecl(inh, use, body) ->
    ObjectL(
      <lower-inherit> inh,
      <map(lower-use)> use,
      body
    )
```

Lowering of the inherit clause is split from this rule, and since the object constructor contains the list of trait uses, we simply map over those using a lowering rule similar to the `lower-inherit` rule. These rules convert the `Inherit(_,_)` and `Use(_,_)` to `InheritL(_,_,_)` and `UseL(_,_,_)` constructors respectively. The main purpose of these rules is that they split the aliasing and exclusion clauses that can appear mixed in the original AST. From this rule it is quite apparent how the transformation is made.

Dynamic semantics The Grace language specification describes object construction as follows (shortened):

> When executed, an object constructor (or trait or class declaration) first creates a new object with no attributes, and binds it to `self`.
>
> Second, the attributes of the superobject (created by the `inherit` clause, possibly modified by `alias` and `exclude`) are installed in the new object.
>
> Third, the methods of all traits are combined.
>
> Fourth, attributes create by local declarations are installed in the new object.
>
> Finally, field initialisers and executable statements are executed. Initialisers for all `def`s and `var`s, and code in the bodies of parents, are executed.

If we consider the DynSem code in Source code 6.3 we can see that the first step of the object construction takes place on line 3. The second and third point from the specification are performed at line 5 and 6. The fourth point from the specification is performed on line 4. The final point is accomplished by line 11. That leaves lines 7 trough 10, these lines are a bit harder to attribute directly to the specification. Line 7 makes a binding of this object, to be used in updating of scopes on line 8 and 9, and the installation of members, on line 10.

Overall, there seems to be a very strong correlation between the informal specification, and the DynSem rule.

Source code 6.3: DynSem rule for object construction.

```
1  S, P Exec() |- ObjectL(inherit, uses, code) --> S'
2  where
3    new-object(S) --> S';
4    snapshot-locals() --> L;
5    S S', O S, P Flatten() |- inherit --> oc-inherit;
6    S S', O S, P Flatten() |- uses --> ocs-use;
7    ObjC(S, src-base(), L, code, oc-inherit, ocs-use, [], []) => oc;
8    read(S') --> Obj(outer, _, slots, methods);
9    update(S', Obj(outer, objc-gather-scopes(oc), slots, methods)) --> _;
10   S' |- install-members-top(oc) --> oc';
11   S' |- init-object(oc') --> U().
```

6.3. Minigrace test suite

On the Minigrace test set, out of 101 tests, Spoofax Grace passes 50 tests. Many of the failing tests can be attributed to the lack of implemented features: operators on numbers, lists, type arguments, *etc.*[3] In the interest of time, these cases are currently not investigated any further.

[3] A notable exception is a test which tests a form of recursion from within an interpolated string (Minigrace: `t120_theBlock`).

However, the main purpose of this project is to make sure the core of the language is supported well, and considering the large test set that is created for that purpose, we feel that this goal has been achieved. Many of the Minigrace tests exhibit features which are not part of the core of the language, so we consider the lower score on this test suite not as a problem. However, it would be nice to also have all these language features, but this would require a larger investment of time.

6.4. Omitted features of Grace

Due to time constraints, not all language features are fully implemented, this section highlights the omitted features.

- Layout-sensitivity; even though trough an extension of SDF3, layout sensitivity may be specified in the grammar, this leads to a grammar which contains many details which obfuscate the context-free essence of the grammar. This was shown in a preliminary investigation by Eduardo Souza [22]. This has shown that is possible to extend the Grace grammar to accommodate the layout-sensitive features. Since the focus of this work is mostly on dynamic semantics, these features were omitted. However, in order to have a grammar that is unambiguous, we make use of the optional semicolons that are allowed by the Grace syntax: Each statement in Grace *may* be concluded with a semicolon. In Spoofax Grace, any statement *must* be concluded with a semicolon, or it will not be terminated. Grace source files that are used in this work are therefore backwards-compatible with the Grace specification, but not necessarily the other way around.
- Fully-featured type system; The type system implemented in this work is very basic, types can be annotated and will be checked; type expressions can be evaluated and the `match` method can be used to check if object conform to a type. More advanced type features in Grace such as type arguments, type expressions, `where` clauses, and interfaces are not included. Programs including these features may work, but no types will be checked for them. Additionally, not all build-in types are complete.
- Static analysis; although there are some simple static analysis being performed during program transformations, the focus of this work is on dynamic semantics, and therefore no type-checking, name-binding or other static checks are performed before execution. This could lead to some invalid Grace programs being executed without error, but no correct Grace programs should be marked as erroneous by the implementation.

In addition to these points, the following features are not implemented in Spoofax Grace: multi-line strings, return type checking, floating point and non-base-10 numbers, match objects, exceptions, `manifest` and `override` annotations, extendible build-in objects, pluggable checkers, lineups (lists), boolean block short circuiting and Unicode character support.

6.5. Performance

The usability of the system is influenced by the speed at which a language can be developed, and how much time it takes to execute the programs of that language. The turnover rate at which language changes can be processed varies depending on what kind of change needs to be performed. The development time of various changes accumulates because each subsequent step in the building process has to be executed as well. This means a change in the syntax requires every build step to be executed, while a change in the dynamic semantics only requires the interpreter to be re-generated and build. An overview of the rough build times can be seen in Table 6.1.[4]

Execution of Grace programs is also relevant for the usability of the system. Generally, in our set of testing programs, the required computation is minimal, and programs are in the order of tens of lines of code. When executing our test-set, the first program which is executed takes significantly longer

[4]These results were achieved with Windows 10 x64, on an Intel i7-6700k (4.2GHz) with 16GB of memory using Oracle JRE 1.8.0-92.

Table 6.1: Build times for Spoofax Grace

Part of implementation	Time (separate) (s)	Time (cumulative) (s)
Syntax	16	91
Transformations	26	75
Dynamic semantics	49	49

(about 1,5 seconds) than each subsequent test (about 0,1 second). This is expected to be because of class loading and JVM compilation that only needs to be performed for the first test, and therefore each subsequent test runs significantly (about 10×) faster.

It takes 45 seconds to run both test suites. The SPT tests take about 15 seconds, and the program tests take about 30 seconds to run.

Running a single trivial Grace program takes a relatively long time because of its class loading and initialisation. This can be offset by running the Grace interpreter continuously with a Nailgun [47] server, this is supported by the generated interpreter.

In comparison to the other Grace implementation, the time required to rebuild Spoofax Grace is generally longer than for other implementations. For hopper, there is no need to rebuild, so this takes no time. Kernan rebuilds very fast, taking about 2,5 seconds to rebuild. This holds for the initial build as well as after a single change to the lexer. After a change to the dynamic semantics, a rebuild took 1,3 seconds.[5] For Minigrace, a full build takes about 163 seconds, however, when a small change is made to the lexer, a rebuild takes roughly 10 seconds and a change to the semantics (code generator) a rebuild took 14 seconds.[6] These build times can be reviewed in Table 6.2.

Table 6.2: Build times for Grace implementations

Implementation	Time (initial) (s)	Time (change lexer) (s)	Time (change semantics) (s)
Spoofax	91	91	49
Hopper	0	0	0
Kernan	2,5	2,5	1,3
Minigrace	163	10	14

In the next chapter, we highlight work related to this project and discuss other Grace implementations.

[5] Achieved on an Intel i7-6700k (4.2GHz) with 16GB of RAM using XBuild version 14.0 and Mono version 4.8.1 on Ubuntu x64 16.04.2.

[6] Achieved on an Intel i7-6700k (4.2GHz) with 16GB of RAM using Node.js version 6.1.0 and NPM version 3.8.6 on Ubuntu x64 16.04.2.

7

Related work

In this chapter a number of similar works and other Grace implementations are highlighted. Some of the specifications discussed here have a more mathematical character, where the specification more formally specified and sometimes accompanied by proofs of certain language properties. Other specifications are more execution-focussed, these are generally specified using in a kind of declarative style that is strict enough to allow for reasonable execution.

7.1. Formalisations

One of the most foundational works in specifying a language in a formalised manner is the work *The definition of Standard ML* by Milner *et al.* [52]. Here a formal semantics is presented in all its fullness: Syntax, static semantics, types and dynamic semantics.

For non object oriented languages, many formal specifications exist. Such as for C [15, 23, 42, 48]. But also for even lower level languages such as ARM [8] and x86 [56] there exist formalised (executable) specifications.

Because Grace is object-oriented, we consider works such as *K-Java: A Complete Semantics of Java* by Bogdanas and Rosu [17]. This work focussus more on the *executable* part of a language specification. They created a complete executable formal semantics of Java 1.4 using the K-framework [55]. The parser-generator used is similar to SDF, as they use a generalised scannerless parser system. The grammar is notated in BNF (Backus-Naur Form) style, rather than the algebraic style SDF uses. A comprehensive test suite is used to validate their semantics. Their approach included splitting up the static and dynamic semantics, and pre-processing Java programs to a certain subset before execution. The semantics were used to model-check multi-threaded programs.

Other formalisations of the Java programming language include: *A Formal Executable semantics for Java* [9], using Centaur [18] and *A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler* [40] using a more tractable Java-derived language to prove a large number of language properties, using the Isabelle framework [63].

In addition to Java, other similar projects exists for other object-oriented programming languages, such as for JavaScript.

Bodin *et al.* present *A Trusted Mechanised JavaScript Specification*, a formalisation of the ECMA standard in the Coq proof assistant, and a reference interpreter for JavaScript extracted from Coq and ported to OCaml [16]. They aim to ensure that their system is an accurate formulation of the ECMA standard.

Guha *et al.* present *The Essence of JavaScript*. They created a core calculus for JavaScript called Lambda-JS [30], and with that a small step operational semantics. Their system mostly concerns with the desugarings that convert JavaScript to a core version called λ_{JS}. Parsing input JavaScript program is achieved through using a hand-crafted parser combinator implemented in Haskell. The desugaring program was implemented with Haskell as well and the semantics for the λ_{JS} sub-language was created with PLT Redex [25]. The system was tested against the Mozilla JavaScript test suite. Finally, they demonstrate this system by implementing some safety features for the languages and providing proofs for showing these programs have certain safety properties.

Other formalisations exist for JavaScript such as *An Operational Semantics for JavaScript* by Maffeis *et al.* [46], however in this case the presented operational semantics are not executable.

Similar works exist for Python, another popular object oriented language. In *Python: The Full Monty* by Politz *et al.* [54] presented a small step operational semantics for Python. This work follows a similar pattern as the formalisation of Guha, where there is a parser and a desugaring program to convert Python programs to a core language (λ_π) and a small step operational semantics for this core language. The parser and desugarer are implemented with Racket [26], and the interpreter for the core language is build using PLT Redex [25]. The system is validated against the CPython [1] test suite. They use the system to highlight certain Python peculiarities.

Many of the aforementioned works regarding object oriented language above follow a similar pattern. For executable specifications, it is common to first translate the language to some sort of core form, and subsequently perform the operational semantics on this core language. This is the same approach that Spoofax Grace follows.

7.2. Grace implementations

There are currently three main other Grace implementations: Minigrace, Kernan and Hopper. In this section the details and origin of these implementation is discussed.

Minigrace is the most widely used implementation of Grace [12]. It was originally created by Michael Homer, but the development has been handed over to Andrew Black after a couple of years. Minigrace is a self-hosted compiler that was bootstrapped using the Parrot Compiler Toolkit [7]. Originally Minigrace was targeting LLVM [43], but currently it is generating both C and JavaScript code. Since the compiler also targets JavaScript, it is able to generate an online IDE that allows users to run Grace programs in the browser.

To parse Grace files, there is a separate lexer which tokenizes the input and passes it to a handwritten recursive descent parser. Since the compiler can be compiled to, and can output JavaScript code, this allows Minigrace to be run in the web browser. A public version of this compiler can be found at: `http://web.cecs.pdx.edu/~grace/ide/`

Although this implementation is very complete and highly functional, due to the browser-interoperability, none of its primary components (parser, transformations, code generator) are formalized, making it very hard to grasp the semantics from it, or to correlate this implementation to the informal Grace language specification. The source code for Minigrace can be found on the following GitHub repository: `https://github.com/gracelang/minigrace/`.

Kernan is an interpreter created by Michael Homer, written in C#. Kernan aims to implement the language in a complete way, however it is not updated very often around the time of writing [32]. Like Minigrace, Kernan has a separate lexer and parser, that architecturally are very similar. The main differences between Minigrace and Kernan is that Kernan is an interpreter rather than a compiler and that Kernan is written in a different language than the target language.

Kernan uses either the Microsoft .NET framework [2] or the open source Mono framework [5], allowing it to run on most platforms. The source code for Kernan can be found in the following git repository: `https://mwh.nz/git/kernan`.

Hopper is an AST interpreter written in JavaScript using Node.js [6] created by Timothy Jones [34]. Hopper it similar to Kernan, it is also an interpreter written in a different language than the target language, and it is quite simple and clean. Also, it has a separate lexer and recursive descent parser. Hopper has not been updated in the past two years, so changes that have been made to the Grace language since then are not supported. Hopper's source code can be retrieved from GitHub: `https://github.com/zmthy/hopper`

These three implementations all use a separate lexer and parser. These parsers and lexers have their true meaning (the Grace grammar) hidden in the implementation; they are embedded in the code. This means it can be quite hard to extract these properties and compare them to other implementations or the informal language specification.

From Minigrace, Kernan and Hopper we can figure out that they all use an imperative style of checking whether a character can be part of an identifier. The Spoofax implementation only declares what kind of characters can be part of the identifier, but completely abstracts over how this is done in the implementation.

Other than the lexer and parser, the actual implantation of the static and dynamic semantics is also embedded in the code. It is very hard to track language features and properties throughout the implementation, although through an initial investigation, this seems to be more easy to do for Kernan and Hopper than for Minigrace, because they do not need to generate code, but rather can operate on an internal model directly instead.

Another beneficial factor for Kernan and Hopper, is that they benefit from better IDE support and language tooling, as they are written in popular general purpose programming languages (C# and JavaScript respectively). Since Minigrace is mostly written in Grace itself, it does not benefit from a great variety of advanced tools.
Spoofax Grace is written in a number of languages (SDF3, Stratego, DynSem, Java), all of which have at least some IDE support in the form of syntax highlighting and basic static analysis, which greatly aids development.

In the next and final chapter we conclude this thesis and discuss possibilities for future work.

8

Discussion

The Grace programming language could benefit from a more formal and executable language specification, in addition to its current informal specification document and other implementations. Spoofax Grace delivers such an implementation at its core, but it is not yet fully complete, as many of the Minigrace implementation tests do not pass, and various features are not implemented. Considering the constraints of this project, it seems appropriate to focus on the core of Grace: validating the crucial parts, and leave other language features as future work.

However Spoofax Grace can form a platform for experimentation, and could be extended to support the full Grace programming language. In addition, the test suite developed for Spoofax Grace can aid other Grace implementation efforts, and steer discussion about the language.

Reverse engineering the semantics of the Grace programming languages was one of the greatest challenges, and now these efforts are cemented in the specification and the tests. A test driven approach proved invaluable here.

Working with the tools the Spoofax language workbench provides was a good experience overall. With the addition of DynSem to the suite of meta-languages, there is now a possibility to develop interpreted languages from A to Z inside a single environment, which lowers the overhead of implementing a language.

8.1. Future work

A number of suggestions for further work are highlighted in the following sections.

8.1.1. Completing Grace features

It would be great to complete Spoofax Grace to support all features of Grace. This includes syntactic features (mostly layout-sensitivity), as well as dynamic features (match objects, exceptions). Even though SDF3 does not include layout-sensitive features, support for this has been added to the language [24]. A preliminary investigation by Eduardo Souza [22] has shown that the layout-sensitive features of Grace can be added to the SDF3 grammar. However it may be argued that this convolutes the grammar to such an extend that it loses its cleanliness. In addition, more desugaring is required to strip away the additional AST noise. As far as other missing (dynamic) features are concerned, there should be no large technical issues for completing these.

8.1.2. Static analysis

Currently the specification does not specify any form of static analysis. The transformation rules can be extended to include a number of static analysis that could report errors back to users before executing any code. These analyses could include: unresolved identifiers, invalid declarations, such as declaring a method in a method, or non-method in a trait, shadowing variables, *etc.* Since Spoofax already has support for reporting issues back to programmers using IDE features, this could be a welcome addition.

On the other hand, it can also be argued that all transformations should be moved into the dynamic domain. This would make the code more centralized, and specified in the same meta-language. However, to accommodate this it would be very useful to have some more basic functional programming features build into DynSem, such as: map, filter, reverse, *etc.*

8.1.3. Setting up a universal Grace test suite

To map the differences between current Grace implementations, a universal test platform could be established. This tool could serve to show the language designers how different implementations handle certain Grace programs, and can help to establish what the desirable behaviour should be.

8.1.4. Exploring Grace performance

Regarding performance of Grace program execution, there are many interesting areas to explore: The performance of current Grace implementation and their relation to the performance of Spoofax Grace, discovering what should be the desired performance for a education targeted language such as Grace, establishing which factors contribute most to the performance in Spoofax Grace (parsing, transforming, *etc.*), and how this related to other Grace implementations.

8.1.5. Making Spoofax Grace more publicly available

Another interesting avenue that could be explored, is the integration of Spoofax Grace with an open (educational) platform such as WebLab (an online learning management system used by TU Delft to manage programming assignments) [39]. This could contribute to the efforts that are already been done with the online Minigrace IDE, but systems like WebLab can also contribute to the deployment of graded assignments, as programs submitted through WebLab are checked and graded on the server side.

8.2. Concluding remarks

Milner *et al.* write in *The definition of Standard ML* [51]:

> "A precise description of a programming language is a prerequisite for its implementation and for its use."

I very much agree with this statement, and to make significant claims about any language feature or property, the formalisation of a language is invaluable.

Grace is an interesting and promising general purpose programming language in the educational sphere, and to have it be supported by a formal specification would aid its practical use and development. The Spoofax language engineering workbench is a good tool for developing an implementation for Grace because it allows the implementation to remain readable and maintainable. The implementation can now serve as a specification as well, or in Milner's words: as a *precise description*. I hope that with this work the foundations for such a precise description has been created, and others are inspired to continue down this path.

Bibliography

[1] Cpython, reference Python implementation. http://python.org/. Accessed: 02-03-2017.

[2] .NET Framework is a software framework developed by Microsoft. https://www.microsoft.com/net. Accessed: 05-04-2017.

[3] The Grace Programming Language. http://gracelang.org/, . Accessed: 29-11-2016.

[4] The Grace Programming Language Draft Specification Version 0.7.7. http://web.cecs.pdx.edu/~grace/doc/gracePdfs/spec_with_grammar.pdf, . Accessed: 05-04-2017.

[5] Mono is an open source implementation of Microsoft's .NET framework. http://www.mono-project.com/. Accessed: 05-04-2017.

[6] Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. https://nodejs.org/. Accessed: 05-04-2017.

[7] Parrot is a virtual machine designed to efficiently compile and execute bytecode for dynamic languages. http://www.parrot.org/. Accessed: 05-04-2017.

[8] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, DAMP '09, pages 13–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-417-1. doi: 10.1145/1481839.1481842. URL http://doi.acm.org/10.1145/1481839.1481842.

[9] Isabelle Attali, Denis Caromel, and Marjorie Russo. A Formal Executable Semantics for Java. In *Proceedings of Formal Underpinnings of Java, an OOPSLA*, volume 98, 1998.

[10] Kent Beck, Erich Gamma, David Saff, and Mike Clark. JUnit is a simple framework to write repeatable tests. http://junit.org/junit4/, 2002. Accessed: 29-11-2016.

[11] Andrew P. Black. Grace grammar. https://github.com/gracelang/language/blob/master/languageSpec/grammar.grace, . Accessed: 05-04-2017.

[12] Andrew P. Black. Minigrace: self-hosting compiler for the Grace programming language. https://github.com/gracelang/minigrace, . Accessed: 29-11-2016.

[13] Andrew P. Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 78–86, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28706. URL http://doi.acm.org/10.1145/28697.28706.

[14] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: The Absence of (Inessential) Difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 85–98, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1562-3. doi: 10.1145/2384592.2384601. URL http://doi.acm.org/10.1145/2384592.2384601.

[15] Sandrine Blazy and Xavier Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. ISSN 1573-0670. doi: 10.1007/s10817-009-9148-3. URL http://dx.doi.org/10.1007/s10817-009-9148-3.

[16] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 87–100, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535876. URL http://doi.acm.org/10.1145/2535838.2535876.

[17] Denis Bogdanas and Grigore Roşu. K-Java: A Complete Semantics of Java. *SIGPLAN Not.*, 50 (1):445–456, January 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676982. URL http://doi.acm.org/10.1145/2775051.2676982.

[18] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. *SIGPLAN Not.*, 24(2):14–24, November 1988. ISSN 0362-1340. doi: 10.1145/64140.65005. URL http://doi.acm.org/10.1145/64140.65005.

[19] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of computer programming*, 72(1): 52–70, 2008.

[20] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[21] Joëlle Despeyroux, Thierry Despeyrout, Laurent Hascoet, and Gilles Kahn. Natural Semantics on the Computer. 1985.

[22] De Souza Amorim Luís Eduardo. Layout sensitive Grace grammar. https://github.com/MetaBorgCube/metaborg-grace/blob/layout-sensitive/grace/syntax/. Accessed: 05-04-2017.

[23] Chucky Ellison and Grigore Rosu. An Executable Formal Semantics of C with Applications. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 533–544. ACM, 2012. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103719.

[24] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. *Layout-Sensitive Generalized Parsing*, pages 244–263. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36089-3. doi: 10.1007/978-3-642-36089-3_14. URL http://dx.doi.org/10.1007/978-3-642-36089-3_14.

[25] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. Mit Press, 2009.

[26] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-80-4. doi: 10.4230/LIPIcs.SNAPL.2015.113. URL http://drops.dagstuhl.de/opus/volltexte/2015/5021.

[27] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language: Everything You Need to Know*. " O'Reilly Media, Inc.", 2008.

[28] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

[29] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390069X, 9780133900699.

[30] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. *The Essence of JavaScript*, pages 126–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14107-2. doi: 10.1007/978-3-642-14107-2_7. URL http://dx.doi.org/10.1007/978-3-642-14107-2_7.

[31] Jan Heering, P. R. H. Hendriks, Paul Klint, and Jan Rekers. The Syntax Definition Formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. doi: 10.1145/71605.71607.

[32] Michael Homer. Kernan is a second-generation reference interpreter for Grace. http://gracelang.org/applications/grace-versions/kernan/. Accessed: 08-03-2017.

[33] Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. Patterns As Objects in Grace. *SIGPLAN Not.*, 48(2):17–28, October 2012. ISSN 0362-1340. doi: 10.1145/2480360.2384581. URL http://doi.acm.org/10.1145/2480360.2384581.

[34] Tim Jones. Hopper is an experimental Grace interpreter written in JavaScript Node.js. http://gracelang.org/applications/grace-versions/hopper/. Accessed: 08-03-2017.

[35] Timothy Jones, Michael Homer, James Noble, and Kim Bruce. Object Inheritance Without Classes. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[36] Gilles Kahn. Natural Semantics. *STACS 87*, pages 22–39, 1987.

[37] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Testing Domain-specific Languages. In Cristina Videira Lopes and Kathleen Fisher, editors, *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 25–26. ACM, 2011. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048160.

[38] Lennart C.L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869497. URL http://doi.acm.org/10.1145/1869459.1869497.

[39] Lennart C.L. Kats, Richard G. Vogelij, Karl Trygve Kalleberg, and Eelco Visser. Software Development Environments on the Web: A Research Agenda. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 99–116, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1562-3. doi: 10.1145/2384592.2384603. URL http://doi.acm.org/10.1145/2384592.2384603.

[40] Gerwin Klein and Tobias Nipkow. A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006. ISSN 0164-0925. doi: 10.1145/1146809.1146811. URL http://doi.acm.org/10.1145/1146809.1146811.

[41] Gabriel D.P. Konat and Jeff Smits. Stratego Manual. http://www.metaborg.org/en/latest/source/langdev/meta/lang/stratego/index.html. Accessed: 12-04-2017.

[42] Robbert Jan Krebbers. *The C standard formalized in Coq*. PhD thesis, Raboud University, 2015.

[43] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977673.

[44] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 2014.

[45] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java® Virtual Machine Specification: Java SE 8 Edition. https://docs.oracle.com/javase/specs/jvms/se8/html/index.html, 2015. Accessed: 29-11-2016.

[46] Sergio Maffeis, John C. Mitchell, and Ankur Taly. *An Operational Semantics for JavaScript*, pages 307–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-89330-1. doi: 10.1007/978-3-540-89330-1_22. URL http://dx.doi.org/10.1007/978-3-540-89330-1_22.

[47] Martian Software, Inc. Nailgun is a client, protocol, and server for running Java programs from the command line without incurring the JVM startup overhead. http://martiansoftware.com/nailgun/. Accessed: 19-04-2017.

[48] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the Depths of C: Elaborating the De Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 1–15, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908081. URL http://doi.acm.org/10.1145/2908080.2908081.

[49] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. ISBN 0-13-247925-7.

[50] Bertrand Meyer. Touch of class. *Learning to program well with Object Technology and Design by Contract, AN INTRODUCTION TO SOFTWARE ENGINEERING*, 2009. URL http://se.inf.ethz.ch/touch.

[51] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1990. ISBN 978-0-262-63132-7.

[52] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, Revised*. MIT Press, Cambridge, MA, USA, 1997.

[53] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

[54] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 217–232, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509536. URL http://doi.acm.org/10.1145/2509136.2509536.

[55] Grigore Roşu and Traian Florin Şerbănută. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010. ISSN 1567-8326. doi: 10.1016/j.jlap.2010.03.012. URL http://www.sciencedirect.com/science/article/pii/S1567832610000160.

[56] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The Semantics of x86-CC Multiprocessor Machine Code. *SIGPLAN Not.*, 44(1):379–391, January 2009. ISSN 0362-1340. doi: 10.1145/1594834.1480929. URL http://doi.acm.org/10.1145/1594834.1480929.

[57] Jeff Smits, Gabriel D.P. Konat, Mark van den Brand, Paul Klint, and Jurgen Vinju. SDF3 reference manual – Spoofax documentation. http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3.html. Accessed: 19-04-2017.

[58] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987. ISSN 0362-1340. doi: 10.1145/38807.38828. URL http://doi.acm.org/10.1145/38807.38828.

[59] Guido Van Rossum et al. Python Programming Language. In *USENIX Annual Technical Conference*, volume 41, page 36, 2007.

[60] Vlad Vergu and Gabriel D.P. Konat. DynSem – Spoofax documentation. http://www.metaborg.org/en/latest/source/langdev/meta/lang/dynsem/index.html. Accessed: 05-04-2017.

[61] Vlad Vergu, Pierre Neron, and Eelco Visser. DynSem: A DSL for Dynamic Semantics Specification. Technical report, Delft University of Technology, Software Engineering Research Group, 2015.

[62] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. Konat. A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014. ISBN 978-1-4503-3210-1. doi: 10.1145/2661136.2661149.

[63] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. *The Isabelle Framework*, pages 33–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-71067-7. doi: 10.1007/978-3-540-71067-7_7. URL http://dx.doi.org/10.1007/978-3-540-71067-7_7.

[64] Christian Wimmer and Thomas Würthinger. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1563-0. doi: 10.1145/2384716.2384723. URL http://doi.acm.org/10.1145/2384716.2384723.

[65] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384587. URL http://doi.acm.org/10.1145/2384577.2384587.

[66] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581. URL http://doi.acm.org/10.1145/2509578.2509581.

A

Grammar in SDF3

Grace

syntax/grace.sdf3

```
module grace

imports

  general
  keyword-rejections
  grace-lowered

context-free start-symbols

  Program
  Exp

context-free syntax

  Program.Program                = <<{Statement ”\n”}*>>

context-free syntax

  Statement.Dialect              = <dialect <STRING>;>
  Statement.Import               = <import <STRING> as <Identifier><Annotations>;>
  Statement.Declaration          = <<Declaration>;>
  Statement.Expression           = <<Exp>;>

context-free syntax

  VarInit.VarInit                = < := <Exp>>
  VarInit.NoVarInit              = <>
  Declaration.Constant           = <def <Identifier><TypeAnn><Annotations>=<Exp>>
  Declaration.Variable           = <var <Identifier><TypeAnn><Annotations><VarInit>>
  Declaration.MethodDecl         = <method <MethodNames><Annotations><TypeRuleRightHand>
  ↪   <MethodBody>>
  Declaration.ClassDecl          = <
                                    class <ClassName><Annotations><TypeRuleRightHand> {
                                        <Inherit><Use*><{Statement ”\n”}*>
                                    }
                                   >
  Declaration.TraitDecl          = <trait <MethodNames><Annotations><TypeRuleRightHand> {
                                        <Use*><{Statement ”\n”}*>
                                    }>
  Use.Use                        = <
                                    use <Exp><Modifier*>;

```

```
                                  >

  Inherit.Inherit                = <
                                   inherit <Exp><Modifier*>;

                                  >
  Inherit.NoInherit              = <>

  Modifier.AliasModifier         = < alias <MethodNames> = <MethodNames>>
  Modifier.ExcludeModifier       = < exclude <MethodNames>>

  ClassName.FullStop             = <<ID>.<MethodNames>>
  ClassName.MethodName           = <<MethodNames>>

  MethodNames.Single             = <<MethodNameNoParam>>
  MethodNames.Multiple           = <<MethodName+>>

  MethodNameNoParam.MethodID     = <<Identifier><TypeArg>>
  MethodNameNoParam.MethodOp     = <<OperatorCF><TypeArg>>

  MethodName.MethodID            = <<Identifier><TypeArg> <Params>>
  MethodName.MethodOp            = <<OperatorCF><TypeArg> <Params>>

  Declaration.TypeDecl          = <type <Identifier><TypeArg> <Annotations> = <TypeDeclBody>>

  TypeDeclBody.TypeDeclBlock     = <<TypeBlock>>
  TypeDeclBody.TypeDeclExp       = <<TypeExp>>

  TypeBlock.TypeBlock            = <{<{TypeRule "\n"}*>}>

  TypeRule.TypeRule              = <<MethodNames> <TypeRuleRightHand>;>
  TypeRuleRightHand.RH           = [ -> [TypeExp]]
  TypeRuleRightHand.NoRH         = <>

//  TypeConf.TypeConf              = [[TypeExp] <: [TypeExp]]
  TypeExp.Union                  = <<TypeExp> + <TypeExp>> {left}
  TypeExp.Subtract               = <<TypeExp> - <TypeExp>> {left}
  TypeExp.Intersect              = <<TypeExp> & <TypeExp>> {left}
  TypeExp.Variant                = <<TypeExp> | <TypeExp>> {left}
  TypeExp.TypeID                 = <<Identifier><TypeArg>>

  TypeExp.AnonType               = <type <TypeBlock>>
  TypeExp.Interface              = <interface <TypeBlock>>

  TypeArg.TypeArg                = <[[<{Identifier ","}*>]]>
  TypeArg.NoTypeArg              = <>

  TypeAnn.TypeAnn                = <: <TypeExp>>
  TypeAnn.NoTypeAnn              = <>

  Identifier.ID                  = <<ID>>
  Identifier.WildCard            = <_>

  Annotations.Annotations        = < is <{Annotation ", "}+>>
  Annotations.NoAnnotations      = <>

  Annotation.Public              = <public>
  Annotation.Readable            = <readable>
  Annotation.Writable            = <writable>
  Annotation.Confidential        = <confidential>
  Annotation.Manifest            = <manifest>
  Annotation.Overrides           = <override>

  Params.Params                  = <(<{Param ", "}+>)>
  Param.ParamWType               = <<Identifier><TypeAnn>>

  MethodBody.MethodBody          = <
                                   {
                                       <{Statement "\n"}*>
                                   }
```

```
                                   >

context-free syntax

  Exp.ObjectDecl                = <
                                   object {
                                       <Inherit><Use*><{Statement ”\n”}*>
                                   }
                                  >

  Exp.TypeExp                   = <<TypeExp>> {avoid}
  Exp.Number                    = <<NUM>>
  Exp.String                    = <<STRING>> {prefer}
  Exp.InterpolatedString        = <<InterpolatedString>>
  Interpolated.IntPol           = <<Exp> <STRINGINTMID>>
  InterpolatedEnd.IntPolEnd     = <<Exp> <STRINGINTEND>>
  Exp.Boolean                   = <<Boolean>>

  InterpolatedString.IntPolStr  = <<STRINGINTSTART><Interpolated*><InterpolatedEnd>>

  Exp                           = <(<Exp>)>                     {bracket}

  Exp.MCallOpEx                 = <<Exp> <OperatorCF> <Exp>>    {left}
  Exp.MCallOpExAssign           = <<Exp> := <Exp>>              {left} // prefer
  Exp.MCallWDot                 = <<Exp>.<Part+>>               {left}
  Exp.MCallImpl                 = <<Part+>>                     {left}

  Exp.Self                      = <self>
  Exp.SelfType                  = <Self>
  Exp.Outer                     = <outer>
  Exp.MQCallOuter               = <<Exp>.outer>                 {left}
  Exp.MQCallSelf                = <<Exp>.self>                  {left}

  Exp.MCallPrefixOpExp          = <<OperatorCF> <Exp>>          {right}

  Exp.Ellipsis                  = <...>
  Exp.LineupExp                 = <<Lineup>>
  Exp.BlockExp                  = <<Block>>

  Exp.Return                    = <return <Exp>>                {right}

  Exp.MatchCase                 = <
                                   match (<Exp>)
                                       <{Case ”\n”}+>
                                  > {prefer} // over methodcall
  Case.Case                     = [case {[CaseExp] [Arrow] [{Statement ”\n”}*]}]
  Case.CaseParen                = [case ( {[CaseExp] [Arrow] [{Statement ”\n”}*]} )]
  CaseExp.CaseExp               = <<CaseLiteral>>
  CaseExp.CaseExpMulti          = <<CaseLiteral> <BoolOp> <CaseLiteral>>
  CaseExp.ExpParens             = <(<Exp>)>
  CaseExp.CIdentifier           = <<Identifier>>
  CaseExp.ExpTyped              = <<Identifier> : <TypeExp>>
  BoolOp.Or                     = <|>
  BoolOp.And                    = <&>
//  CaseType.Type                 = <<Identifier>>
  CaseLiteral.CNumber           = <<NUM>>
  CaseLiteral.CString           = <<STRING>>
  CaseLiteral.CBoolean          = <<Boolean>>

  Arrow.Ascii                   = [->]

context-free syntax

  Part.Part                     = <<Identifier><CallArgs>>

  CallArgs.ArgsParen            = < (<{Exp ”, ”}+>)>            {left}
  CallArgs.NoArgs               = <>

  CallArgs.ArgBlock             = < <Block>>
  CallArgs.ArgNumber            = < <NUM>>
```

```
  CallArgs.ArgString              = < <STRING>>
  CallArgs.ArgInterpolatedString= < <InterpolatedString>>
  CallArgs.ArgLineup              = < <Lineup>>
  CallArgs.ArgBoolean             = < <Boolean>>

  OperatorCF.OperatorCF           = <<Operator>>

  Boolean.True                    = <true>
  Boolean.False                   = <false>
  Lineup.Lineup                   = <[<{Exp ", "}*>]>
  Block.Block                     = <{<{Statement "\n"}*>}>
  Block.BlockWParams              = <{<BlockParams> <{Statement "\n"}*>}> {prefer}
  BlockParams.BlockParams         = [[{Param ", "}*] ->]

context-free priorities

  Exp.MCallWDot > Exp.Return
  ,
  Exp.MCallWDot >
  Exp.MCallPrefixOpExp >
  Exp.MCallOpEx >
  Exp.MCallOpExAssign >
  Exp.TypeExp
  ,
  Exp.MQCallOuter >
  Exp.MCallOpExAssign
  ,
  Exp.MQCallSelf >
  Exp.MCallOpExAssign
  ,
  Exp.MQCallOuter >
  Exp.MCallOpEx
  ,
  Exp.MQCallSelf >
  Exp.MCallOpEx
  ,
  {left: TypeExp.Union
  TypeExp.Subtract
  TypeExp.Intersect
  TypeExp.Variant }

template options

  tokenize : "."
  tokenize : "("
  tokenize : ")"
  tokenize : "{"
  tokenize : "}"
```

syntax/general.sdf3

```
module general

lexical syntax

  ID              = [a-zA-Z] [a-zA-Z0-9\'\_]* Assign?
  ID              = PrefixOperator
  Assign          = ":="
  NUM.Integer     = [1-9][0-9]* // prefer neg int over neg operator
  NUM.IntegerZ    = "0" // prefer neg int over neg operator
  NUM.Decimal     = [0-9]* "." [0-9]+
  NUM.RadixNum    = [02-9][0-9]* [xX] [a-zA-Z0-9]+ // some radix number
  NUM.RadixNum2   = [1][0-9]+ [xX] [a-zA-Z0-9]+ // some radix number
  NUM.SciNum      = [0-9]* "." [0-9]+ [eE] "-"? [0-9]+ // scientific notation
  NUM.SciNum2     = [0-9]+ [eE] "-"? [0-9]+
```

```
  PrefixOperator = "prefix" Operator
  Operator       = [\!\?\@\#\%\^\&\|\~\=\+\-\*\/\\\>\<\:\.\$]+
  STRING         = "\"" StringChar* "\""
  STRINGINTSTART = [\"] StringChar* [\{]
  STRINGINTMID   = [\}] StringChar* [\{]
  STRINGINTEND   = [\}] StringChar* [\"]
  StringChar     = ~[\"\n\{\}]
  StringChar     = "\\\""
  StringChar     = BackSlashChar [\{]
  StringChar     = BackSlashChar [\}]
  StringChar     = BackSlashChar
  BackSlashChar  = "\\"
  LAYOUT         = [\ \n\r] // tabs are not allowed!
  LAYOUT         = "//" ~[\n\r]* NewLineEOF
  NewLineEOF     = [\n\r]
  NewLineCR      = [\r]
  NewLineCR      = [\n\r]
  NewLineEOF     = EOF
  EOF            =

lexical restrictions

  // Ensure greedy matching for lexicals

  NUM           -/- [a-zA-Z0-9\_]
  ID            -/- [a-zA-Z0-9\_]

  Operator      -/- [\!\?\@\#\%\^\&\|\~\=\+\-\*\/\\\>\<\:\.\$]

  // EOF may not be followed by any char

  EOF           -/- ~[]

  // Backslash chars in strings may not be followed by doublequote

  BackSlashChar -/- [\"]

context-free restrictions

  // Ensure greedy matching for comments

  LAYOUT? -/- [\ \n\r]
  LAYOUT? -/- [\/].[\/]
```

syntax/keyword-rejections.sdf3

```
module keyword-rejections

imports

  general

lexical syntax

  ID = "alias" {reject}
  ID = "as" {reject}
  ID = "class" {reject}
  ID = "def" {reject}
  ID = "dialect" {reject}
  ID = "exclude" {reject}
  ID = "import" {reject}
  ID = "inherit" {reject}
  ID = "is" {reject}
  ID = "method" {reject}
  ID = "object" {reject}
```

```
  ID = ”outer” {reject}
  ID = ”prefix” {reject}
  ID = ”required” {reject}
  ID = ”return” {reject}
  ID = ”self” {reject}
  ID = ”Self” {reject}
  ID = ”trait” {reject}
  ID = ”type” {reject}
  ID = ”use” {reject}
  ID = ”var” {reject}
  ID = ”where” {reject}

  ID = ”true” {reject}
  ID = ”false” {reject}

  Operator = ”.” {reject}
  Operator = ”...” {reject}
  Operator = ”:=” {reject}
  Operator = ”=” {reject}
  Operator = ”;” {reject}
  Operator = ”{” {reject}
  Operator = ”}” {reject}
  Operator = ”[” {reject}
  Operator = ”]” {reject}
  Operator = ”(” {reject}
  Operator = ”)” {reject}
  Operator = ”:” {reject}
  Operator = ”->” {reject}

```

Grace-lowered

syntax/grace-lowered.sdf3

```
module grace-lowered

imports

  general
  keyword-rejections
  grace

context-free syntax

  Exp.MCallL            = <_impl (<Identifier>(<{Exp ”, ”}*>))> {prefer}
  Exp.MCallRecvL        = <_recv (<Exp>).<Identifier>(<{Exp ”, ”}*>)> {prefer}
  Exp.ObjectL           = <
                            _object {
                              <Inherit><Use*><{Statement ”\n”}*>
                            }
                            > {prefer}
  Inherit.InheritL      = <_inherit <Exp><Alias*><Exclude*>;>
  Use.UseL              = <_use <Exp><Alias*><Exclude*>;>
  Alias.AliasL          = <_alias <Identifier> = <Identifier>>
  Exclude.ExcludeL      = <_exclude <Identifier>>

  Exp.BlockL            = [_block { [Identifier*] | [{TypeExp ”, ”}*] -> [{Statement ”\n”}*]
  ↪  }] {prefer}
  Exp.Uninitialized     = <_uninit> {prefer}
  TypeExp.Unkwn         = <_Unkwn> {prefer}
                                // methodname      typearguments    annotations     formal
  ↪  argument names    formal argument types returntype
  Declaration.MethodL   = [
                            _method [Identifier] ||[Identifier*]|| [Annotations] ([{Identifier
  ↪  ”, ”}*]) : ([{TypeExp ”, ”}*]) -> [TypeExp] {
```

```
                              [{Statement "\n"}*]
                            }
                            ] {prefer}
  Declaration.ConstantL = <_def <Identifier> <TypeExp><Annotations> = <Exp>> {prefer}
  Declaration.VariableL = <_var <Identifier> <TypeExp><Annotations> := <Exp>> {prefer}

  TypeRule.TypeRuleL    = <_tr <Identifier> ||<Identifier*>|| (<{TypeExp ", "}*>) <TypeExp>>
```

B

Program transformations in Stratego

Desugaring

trans/desugar/desugar.str

```
module trans/desugar/desugar

imports

  src-gen/signatures/grace-sig
  src-gen/signatures/grace-lowered-sig
  src-gen/signatures/general-sig

  trans/desugar/common
  trans/desugar/matchcase
  trans/desugar/lower
  trans/desugar/unquote
  trans/desugar/interpolate
  trans/desugar/annotate
  trans/desugar/analyse

rules

  desugar-pre =
    topdown(analyse-traits)
    ; desugar-program
    <+ desugar-fail

  desugar-all = innermost(desugar)

  desugar =
    desugar-class-declaration
    <+ desugar-trait-declaration
    <+ desugar-optimize-annotations
    <+ desugar-annotate-defaults
    <+ desugar-annotations
    <+ desugar-missing-return-types
    <+ desugar-missing-annotated-types
    <+ desugar-match-case
    <+ desugar-methodOp
    <+ desugar-flatten-methodID
    <+ desugar-flatten-methodPart
    <+ desugar-mcallopexp
    <+ desugar-mcallopexpassign
    <+ desugar-mcallprefixopexp
    <+ desugar-block
    <+ desugar-arg-noargs
    <+ desugar-arg-argsparen
```

```
  <+ desugar-argBlock
  <+ desugar-flatten-objectdecl
  <+ desugar-flatten-declaration
  <+ desugar-unquote-strings
  <+ desugar-interpolate

desugar-all-post: ast -> <lower-post-all> <lower-all> ast

desugar-program: Program(code) ->
  Program([Expression(ObjectDecl(NoInherit(), [], code))])

desugar-missing-return-types: NoRH() -> RH(TypeID(ID(”Unknown”), NoTypeArg()))

desugar-missing-annotated-types: NoTypeAnn() ->
  TypeAnn(TypeID(ID(”Unknown”), NoTypeArg()))

desugar-class-declaration:
  ClassDecl(MethodName(mIDs), annotations, type, inh, use, code) ->
    MethodDecl(
      mIDs,
      annotations,
      type,
      MethodBody([
        Expression(ObjectDecl(inh,use,code))
      ])
    )

desugar-trait-declaration:
  TraitDecl(mIDs, annotations, type, use, code) ->
    MethodDecl(
      mIDs,
      annotations,
      type,
      MethodBody([
        Expression(ObjectDecl(NoInherit(),use,code))
      ])
    )

desugar-class-declaration:
  ClassDecl(FullStop(iden, mIDs), annotations, type, inh, use, code) ->
    Constant(iden, NoTypeAnn(), Annotations([Public()]), Expression (
      ObjectDecl(NoInherit(), [], [
        MethodDecl(
          mIDs,
          annotations,
          type,
          MethodBody([
            Expression(ObjectDecl(inh,use,code))
          ])
        )
      ])
    ))


desugar-methodOp: MethodOp(OperatorCF(n), typeArg, params) -> MethodID(ID(n), typeArg, params)

desugar-flatten-methodID: [ MethodID(n, typeArg, Params(ps)) ] ->
    [ MethodID(ID(name), typeArg, Params(ps)) ]
  where
    n’ := <name-to-string> n;
    name := <concat-strings> [n’, <param-string> ps];
    <not-substring(!”(”)> n’

desugar-flatten-methodID: [ MethodID(n1, typeArg1, Params(p1)),
                            MethodID(n2, typeArg2, Params(p2)) | mids ] ->
    [MethodID(ID(name), types, Params(ps)) | mids ]
  where
    n1’ := <name-to-string> n1;
    n2’ := <name-to-string> n2;
```

```
      types := <merge-typeargs> [typeArg1, typeArg2];
      name := <concat-strings> [n1’, <param-string> p1, n2’, <param-string> p2];
      ps   := <concat> [p1,p2];
      <not-substring(!”(”)> n1’

  desugar-flatten-methodID: [ MethodID(n1, typeArg1, Params(p1)),
                              MethodID(n2, typeArg2, Params(p2)) | mids ] ->
      [MethodID(ID(name), types, Params(ps)) | mids ]
    where
      n1’ := <name-to-string> n1;
      n2’ := <name-to-string> n2;
      types := <merge-typeargs> [typeArg1, typeArg2];
      name := <concat-strings> [n1’, n2’, <param-string> p2];
      ps   := <concat> [p1,p2]

  merge-typeargs: [ TypeArg(ls) , TypeArg(ls2) ] -> TypeArg(<concat> [ls,ls2])
  merge-typeargs: [ NoTypeArg() , TypeArg(ls2) ] -> TypeArg(ls2)
  merge-typeargs: [ TypeArg(ls) , NoTypeArg()  ] -> TypeArg(ls)
  merge-typeargs: [ NoTypeArg() , NoTypeArg()  ] -> NoTypeArg()

  not-substring(s) = is-substring(s) < fail + id

  desugar-flatten-methodPart: [Part(n, NoArgs())] -> <fail>
  desugar-flatten-methodPart: [Part(n, a@_)] -> [Part(ID(name), a)]
    where
      n’ := <name-to-string> n;
      name := <concat-strings> [n’, <param-string> a];
      <is-argument> a;
      // only if n does not contain (
      <not-substring(!”(”)> n’
  desugar-flatten-methodPart: [Part(n1, args1), Part(n2, args2) | ps ] ->
      [Part(ID(name), ArgsParen(args)) | ps]
    where
      n1’ := <name-to-string> n1;
      n2’ := <name-to-string> n2;
      name := <concat-strings> [n1’, <param-string> args1, n2’, <param-string> args2];
      args := <flatten-list> <map(desugar-arg)> [args1, args2];
      // make sure to process first list only once.
      <not-substring(!”(”)> n1’

  desugar-flatten-methodPart: [Part(n1, args1), Part(n2, args2) | ps ] ->
      [Part(ID(name), ArgsParen(args)) | ps]
    with
      n1’ := <name-to-string> n1;
      n2’ := <name-to-string> n2;
      name := <concat-strings> [n1’, n2’, <param-string> args2];
      args := <flatten-list> <map(desugar-arg)> [args1, args2]

  param-string: ps ->
    <concat-strings> <flatten-list> <concat> [ [”(”] , <commas> <map(!”_”)> ps , [”)”]]
      where
        <is-list> ps

  param-string: a@ArgsParen(ps) -> <param-string> ps
  param-string: a@p -> ”(_)”

  commas: [] -> []
  commas: [ a | [] ] -> [ a ]
  commas: [ a | as ] -> [ a , ”,” | <commas> as]

  neq(|a,b) = equal(|a, b) < fail + id

  is-argument: a -> a
    where
       <neq(|a, NoArgs())> a

  desugar-mcallopexp: MCallOpEx(recv, OperatorCF(name), arg)->
    MCallWDot(recv, [Part(ID(<concat-strings> [name, ”(_)”]),
      ArgsParen([arg])
    )])
```

```
desugar-mcallopexpassign: MCallOpExAssign(MCallWDot(recv, [Part(ID(name), NoArgs())]),
↪ arg) ->
  MCallWDot(recv, [Part(ID(<concat-strings> [name, ":=(_)"]),
    ArgsParen([arg])
  )])

desugar-mcallopexpassign: MCallOpExAssign(MCallImpl([Part(ID(name), NoArgs())]), arg) ->
  MCallImpl([Part(ID(<concat-strings> [name, ":="]), ArgsParen([arg]))])

desugar-mcallprefixopexp: MCallPrefixOpExp(OperatorCF( op ), arg ) ->
  MCallWDot(arg, [Part(ID(<concat-strings> ["prefix", op]), NoArgs())])

desugar-flatten-objectdecl: ObjectDecl(a, b, xs) -> ObjectDecl(a, b, ys)
  where
    ys := <flatten-list> xs;
    <not(eq)> (xs, ys)

desugar-flatten-declaration: Declaration([a, b]) ->
    [Declaration(a), Declaration(b)]

desugar-arg:  ArgNumber(a) -> Number(a)
desugar-arg: ArgBoolean(a) -> Boolean(a)
desugar-arg:  ArgString(a) -> String(a)
desugar-arg:  ArgLineup(a) -> LineupExp(a)
desugar-arg: ArgInterpolatedString(intpolstr) -> InterpolatedString(intpolstr)
desugar-arg:  ArgsParen(a) -> a
desugar-arg:            a -> a

desugar-arg-noargs: ArgsParen(as) -> ArgsParen(as')
  where
    as' := <flatten-list> <filter-no-args> as;
    <not(eq)> (as, as')

filter-no-args: [NoArgs() | as] -> [<filter-no-args> as]
filter-no-args: [a | as] -> [a | <filter-no-args> as]
filter-no-args: [] -> []

desugar-arg-argsparen: ArgsParen(as@[_ | _]) ->
  ArgsParen( <flatten-list> <map(strip-argsparen)> as)

strip-argsparen: ArgsParen(a) -> a

desugar-argBlock: ArgBlock(o) -> ArgsParen([BlockExp(o)])

desugar-block:
  Block(BlockWParams(a, b)) -> BlockWParams(a, b)

desugar-block:
  Block(a) -> BlockWParams(BlockParams([]), a)

desugar-fail: a -> <fail>
```

Lowering

trans/desugar/lower.str

```
module trans/desugar/lower

imports

  src-gen/signatures/grace-sig
  src-gen/signatures/grace-lowered-sig
  src-gen/signatures/general-sig
```

```

  trans/desugar/common
  trans/desugar/analyse

rules

  lower-all = innermost(lower)

  lower = lower-mdecl <+
          lower-methodcallwdot <+
          lower-mcallopex <+
          lower-mcallimpl <+
          lower-mcallwdot <+
          lower-objectdecl <+
          lower-constant <+
          lower-variable <+
          lower-block <+
          lower-type-unknown <+
          lower-typerule <+
          lower-fail

  lower-post-all: ast -> <topdown(lower-post-analyse)> <topdown(try(lower-post-2))> <topdown(try(lower-post-1))> ast

  lower-post-1 =
          flatten-statements-declaration <+
          lower-fail

  lower-post-2 =
          flatten-statements <+
          lower-fail

  lower-post-analyse = id

  lower-methodcallwdot:
    MCallWDot(recv, [Part(idf, args)]) ->
      MCallRecvL(recv, <name-to-id> idf, as)
        with
          as := <flatten-list> <lower-arguments> args

  lower-mcallopex:
    MCallOpEx(recv, name, arg) ->
      MCallRecvL(recv, <name-to-id> name, [arg])

  lower-mcallimpl:
    MCallImpl([Part(name , args)]) -> MCallL(name, as)
      with
        as := <flatten-list> <lower-arguments> args

  lower-mdecl:
    MethodDecl(Multiple([MethodID(mName, typeArgs, ps)]), annotations, RH(te), MethodBody(cs)) ->
      MethodL(name, ta, annotations, params, pt, te, cs)
      with
        pt := <lower-get-param-types> ps;
        ta := <lower-get-typeargs> typeArgs;
        params := <lower-get-param-names> ps;
        name := <name-to-id> mName

  lower-mdecl:
    MethodDecl(Multiple([MethodOp(mName, typeArgs, ps)]), annotations, RH(te), MethodBody(cs)) ->
      MethodL(name, ta, annotations, params, pt, te, cs)
      with
        pt := <lower-get-param-types> ps;
        ta := <lower-get-typeargs> typeArgs;
        params := <lower-get-param-names> ps;
        name := <name-to-id> mName

  lower-mdecl:
```

```
    MethodDecl(Single(MethodID(mName, typeArgs)), annotations, RH(te), MethodBody(cs)) ->
      MethodL(<name-to-id> mName, ta, annotations, [], [], te, cs)
      with
        ta := <lower-get-typeargs> typeArgs

  lower-get-typeargs: TypeArg(tas) -> tas
  lower-get-typeargs: NoTypeArg() -> []

  lower-mcallwdot:
    MCallWDot(recv, [Part(ID(name), args)]) ->
      MCallRecvL(recv, name, <lower-arguments> args)

  lower-block: BlockExp(blk) -> blk
  lower-block: BlockWParams(BlockParams(ps), cs) -> BlockL(params, types, cs)
      with
        params := <lower-get-param-names> ps;
        types := <lower-get-param-types> ps

  lower-objectdecl: ObjectDecl(a, b, c) -> ObjectL(<lower-inherit> a, <map(lower-use)> b, c)

  lower-inherit: Inherit(exp, mods) -> InheritL(exp, alias, exclude)
    with
      alias := <get-lowered-alias> mods;
      exclude := <get-lowered-exclude> mods

  lower-inherit: noi@NoInherit() -> noi

  lower-use: Use(exp, mods) -> UseL(exp, alias, exclude)
    with
      alias := <get-lowered-alias> mods;
      exclude := <get-lowered-exclude> mods

  get-lowered-alias: [] -> []
  get-lowered-alias: [AliasModifier(Single(MethodID(toId, _)), Single(MethodID(fromId, _))) | as]
    -> [AliasL(toId,fromId) | <get-lowered-alias> as]
  get-lowered-alias: [AliasModifier(Multiple([MethodID(toId, _, _)]), Multiple([MethodID(fromId, _, _)])) | as]
    -> [AliasL(toId,fromId) | <get-lowered-alias> as]
  get-lowered-alias: [a | as] -> <get-lowered-alias> as

  get-lowered-exclude: [] -> []
  get-lowered-exclude: [e@ExcludeModifier(Single(MethodID(iden, _))) | es]
    -> [ExcludeL(iden) | <get-lowered-exclude> es]
  get-lowered-exclude: [e@ExcludeModifier(Multiple([MethodID(iden, _, _)])) | es]
    -> [ExcludeL(iden) | <get-lowered-exclude> es]
  get-lowered-exclude: [e | es] -> <get-lowered-exclude> es

  lower-constant: Constant(a, t, b, c) -> ConstantL(a, <lower-typeann> t, b, c)

  lower-variable: Variable(a, t, b, NoVarInit()) ->
    VariableL(a, <lower-typeann> t, b, Uninitialized())
  lower-variable: Variable(a, t, b, VarInit(exp)) ->
    VariableL(a, <lower-typeann> t, b, exp)

  lower-typeann: TypeAnn(t) -> t

  lower-type-unknown: TypeID(ID("Unknown"), NoTypeArg()) -> Unkwn()

  lower-typerule: TypeRule(Single(MethodID(iden, typeArg)), RH(retType)) ->
    TypeRuleL(iden, <lower-get-typeargs> typeArg, [], retType)
  lower-typerule:
    TypeRule(Multiple([ MethodID(iden, typeArgs, ps)]), RH(retType)) ->
      TypeRuleL(iden, <lower-get-typeargs> typeArgs, <lower-get-param-types> ps, retType)

  flatten-statements-declaration: Declaration([a,b]) -> [Declaration(a),Expression(b)]

  flatten-statements: ObjectL(a, b, code) -> ObjectL(a, b, <flatten-list> code)
  flatten-statements: MethodL(n, ta, a, p, pt, t, code) ->
    MethodL(n, ta, a, p, pt, t, <flatten-list> code)
```

```
  flatten-statements: BlockL(p, t, code) -> BlockL(p, t, <flatten-list> code)

  lower-get-param-names: Params(ps) -> <lower-get-param-names> ps
  lower-get-param-names: [] -> []
  lower-get-param-names: [ParamWType(n, _)] -> [n]
  lower-get-param-names: [ParamWType(n, _) | bs] -> [n | <lower-get-param-names> bs ]

  lower-get-param-types: Params(ps) -> <lower-get-param-types> ps
  lower-get-param-types: [] -> []
  lower-get-param-types: [ParamWType(_, TypeAnn(te))] -> [te]
  lower-get-param-types: [ParamWType(_, TypeAnn(te)) | ps] ->
    [te | <lower-get-param-types> ps ]

  lower-arguments: ArgNumber(a) -> [Number(a)]
  lower-arguments: ArgString(a) -> [String(a)]
  lower-arguments: ArgBoolean(a) -> [Boolean(a)]
  lower-arguments: ArgLineup(a) -> [LineupExp(a)]
  lower-arguments: ArgInterpolatedString(a) -> [InterpolatedString(a)]
  lower-arguments: NoArgs() -> []
  lower-arguments: ArgsParen(as) -> <lower-arguments> as
  lower-arguments: [ArgsParen(a) | b] -> [a | <lower-arguments> b]
  lower-arguments: a -> a

  lower-fail: a -> <fail>
```

Auxiliary transformations

trans/desugar/annotate.str

```
module trans/desugar/annotate

imports

  src-gen/signatures/grace-sig
  src-gen/signatures/general-sig

rules

  desugar-annotate-defaults: Constant(nm, typeAnn, NoAnnotations(), exp)
    -> Constant(nm, typeAnn, Annotations([Confidential()]), exp)

  desugar-annotate-defaults: Variable(nm, typeAnn, NoAnnotations(), init)
    -> Variable(nm, typeAnn, Annotations([Confidential()]), init)

  desugar-annotate-defaults: MethodDecl(nm, NoAnnotations(), rh, body)
    -> MethodDecl(nm, Annotations([Public()]), rh, body)

  desugar-annotate-defaults: ClassDecl(nm, NoAnnotations(), t, inh, use, code)
    -> ClassDecl(nm, Annotations([Public()]), t, inh, use, code)

  desugar-annotate-defaults: Import(f, nm, NoAnnotations())
    -> Import(f, nm, Annotations([Confidential()]))

  desugar-annotate-defaults: TypeDecl(name, ta, NoAnnotations(), tb)
    -> TypeDecl(name, ta, Annotations([Public()]), tb)

  desugar-annotations: Constant(nm, typeAnn, Annotations([Public()]), exp)
    -> Constant(nm, typeAnn, Annotations([Readable()]), exp)

  desugar-annotations: Variable(nm, typeAnn, Annotations([Public()]), init)
    -> Variable(nm, typeAnn, Annotations([Readable(), Writable()]), init)

  desugar-annotations: Import(f, nm, Annotations([Public()]))
    -> Import(f, nm, Annotations([Readable()]))
```

```

  desugar-annotations: MethodDecl(nm, Annotations(anns), rh, body)
    -> MethodDecl(nm, Annotations([Public()]), rh, body)
    where
      <not(elem)> (Confidential(), anns);
      <not(elem)> (Public(), anns)

  desugar-optimize-annotations: Annotations(as)
    -> Annotations(<optimize-annotations> as)
    with
     <check-annotations> as

  optimize-annotations: [] -> []

  optimize-annotations: [anns] -> [anns’]
    where
      <elem> (Public(), anns);
      <elem> (Readable(), anns);
      anns’ := <remove-all(?Readable())> anns

  optimize-annotations: [anns] -> [anns’]
    where
      <elem> (Public(), anns);
      <elem> (Writable(), anns);
      anns’ := <remove-all(?Writable())> anns

  check-annotations: anns -> anns
    where
      <not(<elem> (Public(), anns) ; <elem> (Confidential(), anns))> anns

```

trans/desugar/common.str

```
module trans/desugar/common

imports

  src-gen/signatures/grace-sig
  src-gen/signatures/grace-lowered-sig
  src-gen/signatures/general-sig

rules

  name-to-string: ID(a) -> a where <is-string> a
  name-to-string: OperatorCF(a) -> a where <is-string> a
  name-to-string: a -> a

  name-to-id: ID(a) -> ID(a)
  name-to-id: OperatorCF(a) -> ID(a) where <is-string> a
```

trans/desugar/interpolate.str

```
module trans/desugar/interpolate

imports

  src-gen/signatures/grace-sig
  src-gen/signatures/general-sig


```

```
rules
  external substring(|begin, end)

  trim-string(|b,e): a -> <substring(|b, <subti> (<string-length> a, e))> a

  desugar-interpolate: Part(nm, ArgInterpolatedString(intpolstr))
    -> Part(nm, ArgsParen([InterpolatedString(intpolstr)]))

  desugar-interpolate:
    InterpolatedString(IntPolStr(beginStr,[],IntPolEnd(exp,endStr)))
      -> MCallWDot(
           MCallWDot(
             String(beginStr’)
           , [Part(ID(”++”), ArgsParen([exp]))]
           )
         , [Part(ID(”++”), ArgsParen([String(endStr’)]))]
         )
      with
        beginStr’ := <trim-string(|1,1)> beginStr;
        endStr’ := <trim-string(|1,1)> endStr

  desugar-interpolate:
    InterpolatedString(IntPolStr(bStr, mids@[IntPol(_, _) | _],
      end@IntPolEnd(eexp,eStr))) ->
        MCallWDot(
          MCallWDot(
            sub
          , [Part(ID(”++”), ArgsParen([eexp]))]
          )
        , [Part(ID(”++”), ArgsParen([String(endStr’)]))]
        )
     where
       IntPol(mExp, mStr) := <last> mids;
       sub := <desugar-interpolate> InterpolatedString(
                        IntPolStr(bStr, <init> mids, IntPolEnd(mExp, mStr)));
       endStr’ := <trim-string(|1,1)> eStr
```

trans/desugar/matchcase.str

```
module trans/desugar/matchcase

imports

  src-gen/signatures/grace-sig
  src-gen/signatures/grace-lowered-sig
  src-gen/signatures/general-sig

  trans/desugar/common

rules

  desugar-case(|liftedname):
    Case(ExpParens(caseExpression), a, body) ->
    <desugar-case(|liftedname, caseExpression)>
      Case(ExpTyped(<new>, TypeID(ID(”Unknown”), NoTypeArg())), a, body)

  desugar-case(|liftedname):
    Case(CaseExp(CString(str)), a, body) ->
    <desugar-case(|liftedname, String(str))>
      Case(ExpTyped(innername, TypeID(ID(”String”), NoTypeArg())), a, body)
  where
    innername := <concat-strings> [”s_”, <new>]

  desugar-case(|liftedname):
    Case(CaseExp(CNumber(num)), a, body) ->
```

```
    <desugar-case(|liftedname, Number(num))>
      Case(ExpTyped(innername, TypeID(ID(”Number”), NoTypeArg())), a, body)
  where
    innername := <concat-strings> [”n_”, <new>]

  desugar-case(|liftedname):
    Case(CaseExp(CBoolean(bool)), a, body) ->
    <desugar-case(|liftedname, Boolean(bool))>
      Case(ExpTyped(innername, TypeID(ID(”Boolean”), NoTypeArg())), a, body)
  where
    innername := <concat-strings> [”b_”, <new>]

  desugar-case(|liftedname):
    Case(CIdentifier(WildCard()), a, body) ->
    <desugar-case(|liftedname)>
      Case(ExpTyped(<concat-strings> [”u_”, <new>], TypeID(ID(”Unknown”), NoTypeArg())), a,
       body)

  desugar-case(|liftedname):
    Case(CIdentifier(iden), a, body) ->
    <desugar-case(|liftedname)>
      Case(ExpTyped(iden, TypeID(ID(”Unknown”), NoTypeArg())), a, body)

  desugar-case(|liftedname):
    Case(ExpTyped(WildCard(), type), a, body) ->
    <desugar-case(|liftedname)> Case(ExpTyped(ID(<new>), type), a, body)

  desugar-case(|liftedname):
    Case(ExpTyped(iden, TypeID(typeId, NoTypeArg())), _, body) ->
    [ BlockExp( Block ( [ Expression (
          MCallWDot(
            BlockExp(
              BlockWParams(
                BlockParams([ParamWType( iden, NoTypeAnn() )])
              , [ Expression(
                  MCallWDot(
                    MCallImpl([Part( typeId , NoArgs())])
                  , [ Part(
                        ID(”match”)
                      , ArgsParen([MCallImpl([Part( iden , NoArgs())])])
                      )
                    ]
                  )
                )
              ]
            )
          )
        , [ Part(
              ID(”apply”)
            , ArgsParen([MCallImpl([Part(ID( liftedname ), NoArgs())])])
            )
          ]
        )
      )]))
    , BlockExp(
        Block( [
          Expression(
            MCallWDot(
             BlockExp(BlockWParams( BlockParams([ParamWType( iden, NoTypeAnn() )]), body )),
              [Part(ID(”apply”), ArgsParen([MCallImpl([Part( liftedname , NoArgs())])]))]
            )
          )
         ] )
      )
    ]

  desugar-case(|liftedname, matchExpr):
    Case(ExpTyped(iden, TypeID(typeId, NoTypeArg())), _, body) ->
    [ BlockExp( Block( [ Expression(
```

```
          MCallWDot(
            MCallWDot(
              MCallImpl([Part( typeId , NoArgs())])
            , [ Part(
                  ID("match(_)")
                , ArgsParen([MCallImpl([Part(ID( liftedname ), NoArgs())])])
                )
              ]
            )
          , [ Part(
                ID("&&(_)")
              , ArgsParen(
                  [ MCallWDot(
                      MCallImpl([Part(ID( liftedname ), NoArgs())])
                    , [Part(ID("==(_)"), ArgsParen([ matchExpr ]))]
                    )
                  ]
                )
              )
            ]
          )

      ) ] ) )
    , BlockExp(
        Block( [
          Expression(
            MCallWDot(
             BlockExp(BlockWParams( BlockParams([ParamWType( iden, NoTypeAnn() )]), body )),
              [Part(ID("apply"), ArgsParen([MCallImpl([Part( liftedname , NoArgs())])]))]
            )
          )
         ] )
      )
    ]

  desugar-case(|liftedname): a -> <fail>
  where
    <debug(!"error: unkown case type: ")> a

  desugar-caseparen-to-case: CaseParen(a,b,c) -> Case(a,b,c)
  desugar-caseparen-to-case: Case(a,b,c) -> Case(a,b,c)

  desugar-match-case:
    MatchCase(matchExpression, cases) ->
      MCallWDot(
        BlockExp(
          BlockWParams(
            BlockParams([ParamWType(ID(liftedname), NoTypeAnn())])
          , [ Expression(
                MCallImpl(
                  [Part(
                    ID(methodName)
                  , ArgsParen(
                      cases''
                    )
                  )]
                )
              )
            ]
          )
        )
      , [Part(ID("apply"), ArgsParen([matchExpression]))]
      )
    where
      liftedname := <concat-strings> ["m_", <new>];
      cases' := <map(desugar-caseparen-to-case)> cases;
      cases'' := <flatten-list> <map(desugar-case(|liftedname))> cases';
      numParts := <length> cases'';
      list := <range(|2)> <int-dec> <int-dec> numParts;
      listNames := <map(!"elseif(_)then(_)")> list;
```

```
    methodName := <concat-strings> ["if(_)then(_)" | listNames]
```

trans/desugar/unquote.str

```
module trans/desugar/unquote

imports

  src-gen/signatures/grace-sig
  src-gen/signatures/general-sig

rules

  desugar-unquote-strings:
    String(s) -> String(<unquote(?'"')> s)

  desugar-unquote-strings:
    ArgString(s) -> ArgString(<unquote(?'"')> s)

  desugar-unquote-strings:
    CString(s) -> CString(<unquote(?'"')> s)

  desugar-unquote-strings:
    Dialect(s) -> Dialect(<unquote(?'"')> s)

  desugar-unquote-strings:
    Import(s, b, c) -> Import(<unquote(?'"')> s, b, c)

  desugar-unquote-strings:
    IntPolEnd(e, s) -> IntPolEnd(e, <unquote(?'"')> s)
```

trans/desugar/analyse.str

```
module trans/desugar/analyse

imports

  src-gen/signatures/grace-sig
  src-gen/signatures/grace-lowered-sig
  src-gen/signatures/general-sig

  trans/desugar/common

rules

  analyse-traits: a@TraitDecl(_,_,_,_,code) -> a
    with
      <only-methods> code
  analyse-traits: a -> a

  only-methods: [] -> []
  only-methods: [Declaration(MethodDecl(_,_,_,_)) | code] ->
    <only-methods> code
  only-methods: a -> <fail> "Only methods declarations may occur in trait."

  analyse-duplicate-decls: a@ObjectL(_,_,code) -> a
    with <no-duplicates> (code, [])
  analyse-duplicate-decls: a@MethodL(_,_,_,ids,_,_,[Expression(ObjectL(_,_,code))]) -> a
    with
      <no-duplicates> (code, <map(name-to-string)> ids)
```

```
analyse-duplicate-decls: a@MethodL(_,_,_,ids,_,_,code) -> a
  with
    <no-duplicates> (code, <map(name-to-string)> ids)

analyse-duplicate-decls: a@BlockL(_,_,code) -> a
  with <no-duplicates> (code, [])
analyse-duplicate-decls: a -> a

no-duplicates: ([],a) -> ([], a)
no-duplicates: ([Declaration(d) | code], names) ->
  <no-duplicates> (code, names')
  with
    names' := <check-duplicate-decls> (d, names);
    <no-dups> names'
no-duplicates: a -> a

// fail if d declares a name already in names, otherwise add name to names
check-duplicate-decls: (MethodL(ID(name),_,_,_,_,_,_), names) -> [name | names]
  with
    <not(elem)> (name, names)
check-duplicate-decls: (ConstantL(ID(name),_,_,_), names) -> [name | names]
  with
    <not(elem)> (name, names)
check-duplicate-decls: (VariableL(ID(name),_,_,_), names) -> [name | names]
  with
    <not(elem)> (name, names)
check-duplicate-decls: (BlockL(ids,_,_), names) -> names
  with
    <not(elem)> (<map(name-to-string)> ids, names)
check-duplicate-decls: (a, names) -> names

no-dups: [] -> []
no-dups: [x|xs] -> <no-dups> xs with <not(elem)> (x,xs)
```

C

Dynamic semantics in DynSem

trans/grace.ds

```
module trans/grace

imports
  trans/semantics/semantics

signature
  arrows
    Program -init-> (V * Exn * H)

rules

  p@Program(_) -init-> (v_out, EX, H)
  where
    next() --> base;
    ProgPath native-term-origin-path(p), R No-Return(), O base, S base, P Exec(), Src src-base() |- p :: H {}, L {},
      VH {}, DCache {}, ICache {}, EX Ok() --> v :: H, L, VH, EX;
    case EX of {
      Ok() =>
        v => v_out
      otherwise =>
        DoneV() => v_out
    }.
```

trans/semantics/semantics.ds

```
module trans/semantics/semantics

imports
  src-gen/ds-signatures/grace-lowered-sig

imports
  trans/semantics/store
  trans/semantics/values
  trans/semantics/objectmodel
  trans/semantics/functions/functions
  trans/semantics/expressions
  trans/semantics/statements
  trans/semantics/numbers
  trans/semantics/booleans
```

```
  trans/semantics/strings
  trans/semantics/imports
  trans/semantics/store

signature
  arrows
    Program --> V

rules

  S |- prog@Program(code) --> v
  where
    load-dialect(collect-dialect-statement(prog)) --> S';
    S', O S |- code --> v.

  [] : Code --> DoneV().

  [c] : Code --> c.

  [c | cs@[_|_]] : Code --> v
  where
    c --> _;
    cs --> v.
```

trans/semantics/controlflow.ds

```
module trans/semantics/controlflow

imports
  src-gen/ds-signatures/grace-lowered-sig
  trans/semantics/expressions

signature
  arrows
    while-loop(Exp, Exp) --> V
    while-loop-evaluated(V, V) --> V

rules

  while-loop(e1, e2) --> v
  where
    e1 --> v1;
    e2 --> v2;
    while-loop-evaluated(v1, v2) --> v.

  w@while-loop-evaluated(v1, v2) --> DoneV()
  where
    call(v1, [], "apply") --> BoolV(cond);
    case cond of {
      true =>
        call(v2, [], "apply") --> _;
        w --> _
      otherwise =>
    }.
```

trans/semantics/expressions.ds

```
module trans/semantics/expressions

imports
```

```
  src-gen/ds-signatures/grace-sig
  trans/semantics/values
  trans/semantics/statements
  trans/semantics/objectmodel

signature
  sorts
    Exn

  constructors
    Ok : Exn
    Exn : String -> Exn

  components
    EX : Exn = Ok()

  arrows
    Exp --> V
    List(Exp) --> List(V)

rules

  Self() --> current-self().

  Outer() --> current-outer().

  Uninitialized() --> UninitializedV().

  [] : List(Exp) --> [].

  [e | es] : List(Exp) --> [ v | vs ]
  where
    e --> v;
    es --> vs.
```

trans/semantics/imports.ds

```
module trans/semantics/imports

imports
  src-gen/ds-signatures/grace-sig
  src-gen/ds-signatures/grace-lowered-sig
  trans/semantics/semantics
  trans/semantics/statements

signature
  sort aliases
    Addr = Int

  components
    DCache : Map(String, Addr)
    ICache : Map(String, V)

  arrows
    collect-dialect-statement(Program) --> Statement
    load-dialect(Statement) --> Addr

    load-import(String) --> V

    parse-file(String) --> Program

    dialect-cache-has(String) --> Bool
    dialect-cache-add(String, Addr) --> Addr
    dialect-cache-get(String) --> Addr
    dialect-path(String) --> String
```

```

    import-cache-has(String) --> Bool
    import-cache-add(String, V) --> V
    import-cache-get(String) --> V
    import-path(String) --> String

  native operators
    native-parse-file: String -> Program
    native-term-origin-path: AST -> String
    native-parent-directory: String -> String
    native-path-separator: String
    native-standardgrace: String

  components
    ProgPath: String

rules

  parse-file(s) --> native-parse-file(s).

  load-dialect(Dialect(”none”)) --> new-object(S)
  where
    current-self() --> RefV(S).

  load-dialect(Dialect(name)) --> dialect-cache-get(name)
  where
    dialect-cache-has(name) --> true.

  load-dialect(Dialect(name)) --> dialect-cache-add(name, dialect)
  where
    dialect-cache-has(name) --> false;
    parse-file(dialect-path(name)) --> program;
    program --> RefV(dialect);
    read(dialect) --> Obj(_, outers, slots, methods);
    update(dialect, Obj(fresh, outers, slots, methods)) --> _.

  collect-dialect-statement(Program([Expression(ObjectL(_,_,[dialect@Dialect(_) | _ ])) | _])) --> dialect.

  collect-dialect-statement(Program([Expression(ObjectL(_,_,[stm | _ ])) | _])) --> Dialect(”standardGrace”)
  where
    stm =!=> Dialect(_).

  collect-dialect-statement(Program([Expression(ObjectL(_,_,[])) | _])) --> Dialect(”standardGrace”).


  load-import(name) --> import-cache-get(name)
  where
    import-cache-has(name) --> true.

  load-import(name) --> import-cache-add(name, import)
  where
    import-cache-has(name) --> false;
    parse-file(import-path(name)) --> program@Program(_);
    program --> import@RefV(_).


  ProgPath |- import-path(name) --> native-parent-directory(ProgPath) ++ native-path-separator() ++ name.

  ProgPath |- dialect-path(name) --> native-parent-directory(ProgPath) ++ native-path-separator() ++ name
  where
    name != ”standardGrace”.

  dialect-path(”standardGrace”) --> native-standardgrace().


```

```
  DCache |- dialect-cache-has(name) --> DCache[name?].

  dialect-cache-add(name, dialect) :: DCache --> dialect :: DCache { name |--> dialect,
  ↪  DCache}.

  dialect-cache-get(name) :: DCache --> DCache[name] :: DCache.


  ICache |- import-cache-has(name) --> ICache[name?].

  import-cache-add(name, import) :: ICache --> import :: ICache {name |--> import, ICache}.

  import-cache-get(name) :: ICache --> ICache[name] :: ICache.

```

trans/semantics/objectmodel.ds

```
module trans/semantics/objectmodel

imports
  src-gen/ds-signatures/grace-sig

imports
  trans/semantics/values
  trans/semantics/store
  trans/semantics/runtime/natives

  trans/semantics/strings
  trans/semantics/numbers
  trans/semantics/lineups
  trans/semantics/statements
  trans/semantics/imports
  trans/semantics/visibility

signature
  sorts
    Addr

  sorts
    Object
    Member

  sort aliases
    Self = Addr
    HeapData = Object
    Slots = Map(Int, V)
    Methods = Map(String, V)

  constructors // extra instructions
    SlotRead : Int -> Statement
    SlotWrite : Int * Exp * TypeExp -> Statement

  constructors
    Obj: Addr * List(Addr) * Slots * Methods -> Object
    RefV: Addr -> V {implicit}

  components
    S : Addr
    O : Addr

rules

  S, P Exec() |- ObjectL(inherit, uses, code) --> S'
  where
    new-object(S) --> S';
```

```
    snapshot-locals() --> L;
    S S', O S, P Flatten() |- inherit --> oc-inherit;
    S S', O S, P Flatten() |- uses --> ocs-use;
    ObjC(S, src-base(), L, code, oc-inherit, ocs-use, [], []) => oc;
    read(S') --> Obj(outer, _, slots, methods);
    update(S', Obj(outer, objc-gather-scopes(oc), slots, methods)) --> _;
    S' |- install-members-top(oc) --> oc';
    S' |- init-object(oc') --> U().

  SlotRead(i) --> ensure-defined(read-slot(i)).

  SlotWrite(i, e, te) --> DoneV()
  where
    e --> v;
    type-check([te], [v]) --> true;
    write-slot(i, v) --> _.






/* ========== OBJECT FLATTENING =========== */
signature
  sorts
    Phase

  constructors
    Exec    : Phase
    Flatten : Phase

  sort aliases
    Source = Int

  components
    P : Phase
    Src : Int

  constructors
    ObjC: Addr * Source * Env * Code * V * List(V) * List(Alias) * List(Exclude) -> V
    NoObjC: V

  arrows
    Inherit --> V
    List(Use) --> List(V)

  arrows
    objc-rec-aliases(V, List(Alias)) --> V
    objc-rec-excludes(V, List(Exclude)) --> V
    objc-gather-scopes(V) --> List(Addr)
    objc-gather-scopes-concat(List(V)) --> List(Addr)

    src-base() --> Source
    src-next() --> Source
    src-previous() --> Source
    src-is-base(Source) --> Bool

rules

  S, P Flatten(), Src |- ObjectL(inherit, uses, code) --> ObjC(S, Src, L, code, oc-inherit,
  ↪ ocs-use, [], [])
  where
    snapshot-locals() --> L;
    O S, P Flatten() |- inherit --> oc-inherit;
    O S, P Flatten() |- uses --> ocs-use.

  P Flatten() |- MCallRecvL(e, ID(x), es) --> v
  where
    P Exec() |- e --> recv;
    P Exec() |- es --> vs;
```

```
    P Flatten() |- call-qualified(recv, x, vs) --> v.

  NoInherit() --> NoObjC().

  InheritL(e, aliases, excludes) --> objc-rec-aliases(objc-rec-excludes(oc, excludes), aliases)
  where
    Src src-next() |- e --> oc@ObjC(_, _, _, _, _, _, _, _).

  [] : List(Use) --> [].

  [UseL(e, aliases, excludes) | uses] : List(Use) --> [oc'|ocs]
  where
    Src src-next() |- e --> oc;
    objc-rec-aliases(objc-rec-excludes(oc, excludes), aliases) --> oc';
    Src src-next() |- uses --> ocs.



  objc-gather-scopes(NoObjC()) --> [].

  objc-gather-scopes(ObjC(O, _, _, _, inherit, uses, _, _)) --> [O | scopes]
  where
    objc-gather-scopes-concat([inherit | uses]) --> scopes.

  objc-gather-scopes-concat([]) --> [].

  objc-gather-scopes-concat([oc | ocs]) --> ocs1 ++ ocs2
  where
    objc-gather-scopes(oc) --> ocs1;
    objc-gather-scopes-concat(ocs) --> ocs2.

  objc-rec-excludes(ObjC(outer, src, L, code, inherit, use, aliases, _), excludes) --> ObjC(outer, src, L, code, inherit, use, aliases, excludes).

  objc-rec-aliases(ObjC(outer, src, L, code, inherit, use, _, excludes), aliases) --> ObjC(outer, src, L, code, inherit, use, aliases, excludes).

  src-base() --> 0.

  Src |- src-next() --> addI(Src, 1).

  Src |- src-previous() --> addI(Src, -1)
  where
    gtI(Src, 0) == true.

  src-is-base(Src) --> eqI(Src, 0).



/* ========== OBJECT MEMBER INSTALLATION ===== */
signature
  sort aliases
    Aliases = List(Alias)
    Excludes = List(Exclude)

  arrows
    install-members-top(V) --> V
    install-members(V) --> V
    install-members-map(List(V)) --> List(V)

    install-code(Code)--> Code

    install-declaration(Declaration) --> Code

    install-import(Statement) --> Code

    install-method(Declaration) --> U

    install-alias(String, V) --> U
```

```
    install-aliases(String, V) --> U

    exclude-method(String) --> Bool

    ensure-unique-method(String, V) --> U

    install-aliases() --> U
    exclude-methods() --> U

  components
    Als: Aliases
    Exs: Excludes

rules

  install-members-top(v) --> v'
  where
    install-members(v) :: NS 0, GS {} --> v'.

  install-members(NoObjC()) --> NoObjC().

  install-members(ObjC(O, Src, L, code, inherit, uses, Als, Exs)) --> ObjC(O, Src, L, code',
  ↪  inherit', uses', Als, Exs)
  where
    install-members(inherit) --> inherit';
    install-members-map(uses) --> uses';
    O, Src |- install-code(code) :: L, Als, Exs --> code' :: L _, Als _, Exs _.

  install-members-map([]) --> [].

  install-members-map([oc | ocs]) --> [oc' | ocs']
  where
    install-members(oc) --> oc';
    install-members-map(ocs) --> ocs'.

  install-code([]) --> []
  where
    install-aliases() --> _;
    exclude-methods() --> _.

  install-code([Declaration(d) | code]) --> decl-code ++ code'
  where
    install-declaration(d) --> decl-code;
    install-code(code) --> code'.

  install-code([imp@Import(_, _, _) | code]) --> imp-code ++ code'
  where
    install-import(imp) --> imp-code;
    install-code(code) --> code'.

  install-code([e | code]) --> [e | code']
  where
    e =!=> Declaration(_);
    e =!=> Import(_, _, _);
    install-code(code) --> code'.

  install-declaration(m@MethodL(_, _, _, _, _, _, _)) --> []
  where
    install-method(m) --> _.

  install-declaration(VariableL(ID(x), type, annos, e)) --> [SlotWrite(i, e, type)]
  where
    add-slot(x) --> i;
    install-method(field-getter(x, i, has-anno-readable(annos))) --> _;
    install-method(field-setter(x, i, has-anno-writable(annos), type)) --> _.

  install-declaration(VariableL(WildCard(), _, _, e)) --> [Expression(e)].

  install-declaration(ConstantL(ID(x), type, annos, e)) --> [SlotWrite(i, e, type)]
  where
```

```
    add-slot(x) --> i;
    install-method(field-getter(x, i, has-anno-readable(annos))) --> _.

  install-declaration(ConstantL(WildCard(), _, _, e)) --> [Expression(e)].

  install-declaration(TypeDecl(ID(x), NoTypeArg(), annos, TypeDeclBlock(tb)))
    --> [SlotWrite(i, TypeExp(AnonType(tb)), Unkwn())]
  where
    add-slot(x) --> i;
    install-method(field-getter(x, i, has-anno-public(annos))) --> _.

  install-import(Import(name, ID(x), annos)) --> []
  where
    add-slot(x) --> i;
    write-slot(i, load-import(name)) --> _;
    install-method(field-getter(x, i, has-anno-readable(annos))) --> _.

  install-method(m@MethodL(ID(x), _, _, _, _, _, _)) --> U()
  where
    method-closure(m) --> clos;
    install-aliases(x, clos) --> _;
    exclude-method(x) --> excluded;
    case excluded of {
      false =>
        ensure-unique-method(x, clos) --> _;
        add-method(x, clos) --> _
      otherwise =>
    }.

  install-aliases(_, _) :: Als [] --> U() :: Als [].

  install-aliases(name, clos) :: Als [a@AliasL(_, ID(x)) | Als] --> u :: Als [a | Als']
  where
    name != x;
    install-aliases(name, clos) :: Als --> u :: Als'.

  install-aliases(name, clos) :: Als [AliasL(ID(x'), ID(x)) | Als] --> u :: Als'
  where
    name == x;
    install-alias(x', clos) --> _;
    install-aliases(name, clos) :: Als --> u :: Als'.

  install-aliases() :: Als [] --> U() :: Als [].

  install-aliases() :: Als [AliasL(ID(x'), ID(x)) | Als] --> U() :: Als []
  where
    disambiguate-closure(lookup-local-method(current-self(), x), x) --> clos;
    install-alias(x', clos) --> _;
    install-aliases() :: Als --> _ :: Als _.

  install-alias(x, clos@ClosV(_, _, _, _, _, _, _, _, _, _, _)) --> U()
  where
    copy-closure(clos) --> clos';
    ensure-unique-method(x, clos') --> _;
    add-method(x, clos') --> _.

  exclude-method(_) :: Exs [] --> false :: Exs [].

  exclude-method(name) :: Exs [ExcludeL(ID(x)) | Exs] --> true :: Exs
  where
    name == x.

  exclude-method(name) :: Exs [e@ExcludeL(ID(x)) | Exs] --> excluded :: Exs [e | Exs']
  where
    name != x;
    exclude-method(name) :: Exs --> excluded :: Exs'.

  exclude-methods() :: Exs [] --> U() :: Exs [].

  exclude-methods() :: Exs [ExcludeL(ID(x)) | Exs] --> U() :: Exs []
```

```
  where
    disambiguate-closure(lookup-local-method(current-self(), x), x) --> _;
    remove-method(x) --> _;
    exclude-methods() :: Exs --> _ :: Exs _.

  ensure-unique-method(x, clos) --> U()
  where
    lookup-local-method(current-self(), x) --> clos';
    case clos' of {
      ClosV(_, _, _, _, _, _, _, _, _, _, _) =>
        closure-source(clos) --> src;
        closure-source(clos') --> src';
        case eqI(src, src') of {
          true =>
            halt-error("Duplicate method: ", x) --> _
          otherwise =>
        }
      otherwise =>
    }.

/* ============== OBJECT INIT ========== */
signature
  arrows
    init-object(V) --> U
    init-object-map(List(V)) --> U

rules

  init-object(NoObjC()) --> U().

  init-object(ObjC(O, _, L, code, inherit, used, _, _)) --> U()
  where
    init-object(inherit) --> _;
    init-object-map(used) --> _;
    O |- code :: L --> _.

  init-object-map([]) --> U().

  init-object-map([oc | ocs]) --> U()
  where
    init-object(oc) --> _;
    init-object-map(ocs) --> _.

/* ========== FIELD METHOD GENERATION ======= */
signature
  arrows
    field-getter(String, Int, Bool) --> Declaration
    field-setter(String, Int, Bool, TypeExp) --> Declaration

  native operators
    mksettername: String -> String

rules

  field-getter(x, i, c) -->
    MethodL(ID(x), [], visibility-annos(c), [], [], no-type(), [SlotRead(i)]).

  field-setter(x, i, c, argType) -->
    MethodL(ID(mksettername(x)), [], visibility-annos(c), [ID("p")],
    [argType], no-type(), [SlotWrite(i, MCallL(ID("p"), [] : List(Exp)), Unkwn())]).

/* ======== OBJECT META-FUNCTIONS ========= */
signature
  sorts
    StatementResult

  constructors
    res: Statement -> StatementResult

  arrows
```

```
    new-object(Addr) --> Addr

    add-slot(String) --> Int

    read-slot(Int) --> V
    write-slot(Int, V) --> U

    add-method(String, V) --> U

    remove-method(String) --> U

    lookup-local-method(V, String) --> V
    lookup-outer-method(V, String) --> V

    current-self() --> V
    current-outer() --> V
    current-method-names() --> List(String)

    is-member(String) --> Bool

    outer(Addr) --> V
    self(Addr) --> V


    identity-check(V) --> V

  components
    NS : Int // NextSlot
    GS : Map(String, Int) // GivenSlots

rules

  S |- current-self() --> S.

  O |- current-outer() --> O.

  self(S) --> S.

  outer(S) --> O
  where
    read(S) --> Obj(O, _, _, _).

  new-object(O) --> S
  where
    allocate(Obj(O, [O], {}, {})) --> S.

  add-slot(x) :: GS --> GS[x] :: GS
  where
    GS[x?] == true.

  S |- add-slot(x) :: NS, GS --> NS :: NS addI(NS, 1), GS {x |--> NS, GS}
  where
    GS[x?] == false;
    read(S) --> Obj(O, outers, slots, methods);
    update(S, Obj(O, outers, {NS |--> UninitializedV(), slots}, methods)) --> _.

  S |- read-slot(i) --> slots[i]
  where
    read(S) --> Obj(_, _, slots, _).

  S |- write-slot(i, v) --> U()
  where
    read(S) --> Obj(O, outers, slots, methods);
    update(S, Obj(O, outers, {i |--> v, slots}, methods)) --> _.

  S |- add-method(x, v) --> U()
  where
    read(S) --> Obj(O, outers, slots, methods);
    update(S, Obj(O, outers, slots, {x |--> v, methods})) --> _.
```

```


  S |- current-method-names() --> allkeys(methods)
  where
    read(S) --> Obj(_, _, _, methods).

  S |- is-member(x) --> methods[x?]
  where
    read(S) --> Obj(_,_,_, methods).



  // Lookup in self
  lookup-local-method(RefV(S'), x) --> v
  where
    read(S') --> Obj(_, _, _, methods);
    case methods[x?] of {
      true =>
        methods[x] => v
      false =>
        DoneV() => v
    }.

  lookup-local-method(v, _) --> DoneV()
  where
    v =!=> RefV(_).

  // Lookup in outers
  lookup-outer-method(RefV(S'), x) --> v
  where
    read(S') --> Obj(O', _, _, methods);
    case methods[x?] of {
      true =>
        methods[x] => v
      false =>
        is-stored(O') --> outer-exists;
        case outer-exists of {
          true =>
            lookup-outer-method(O', x) --> v
          otherwise =>
            DoneV() => v
        }
    }.

  lookup-outer-method(v, _) --> DoneV()
  where
    v =!=> RefV(_).



  identity-check(other) --> BoolV(true)
  where
    other => RefV(addr);
    current-self() == addr.

  identity-check(other) --> BoolV(false)
  where
    other => RefV(addr);
    current-self() != addr.


signature
  arrows
     log-object-creation(Addr) --> Addr

rules

  log-object-creation(S) --> S
  where
    read(S) --> Obj(O, outers, _, methods);
```

```
    concat(separate-by(allkeys(methods), ", ")) --> method-names;
    log("S: " ++ int2string(S : Int) ++ " O " ++ int2string(O : Int) ++
      " outers: " ++ str(outers : AST) ++  ", method-names: " ++ method-names
      ++ ", methods: " ++ str(methods : AST)) --> _.

```

trans/semantics/functions/calls.ds

```
module trans/semantics/functions/calls

imports
  src-gen/ds-signatures/grace-sig
  trans/semantics/statements
  trans/semantics/functions/locals

signature
  sorts
    Return-Marker

  constructors
    //      self    outer   params                                body    env    public source
    ↪ return          paramtypes ret-type
    ClosV: Addr * Addr * List(Identifier) * List(Identifier) * Code * Env * Bool * Source *
    ↪ Return-Marker * List(TypeExp) * TypeExp -> V

    No-Return: Return-Marker
    Return-To: Int -> Return-Marker

    Rex: Int * V -> Exn

  components
    R: Return-Marker

/* =========== call resolution and dispatch =========== */
signature
  arrows
    call-implicit(String, List(V)) --> V
    call-qualified(V, String, List(V)) --> V
    call(V, List(V), String) --> V

    access-local(String, List(V)) --> V

    disambiguate-closure(V, String) --> V
    disambiguate-closure(V, V, String) --> V

    closure-source(V) --> Source

rules

  call-implicit(x, vs) --> access-local(x, vs)
  where
    is-local(x) --> true.

  call-implicit(x, vs) --> call(clos, vs, x)
  where
    is-local(x) --> false;
    lookup-local-method(current-self(), x) --> local-clos;
    lookup-outer-method(current-outer(), x) --> outer-clos;
    log("disambiguate-closure, from implicit call") --> _;
    disambiguate-closure(local-clos, outer-clos, x) --> clos.

  access-local(x, [v]) --> DoneV()
  where
    str-ends-with(x, ":=(_)") --> true;
    update-local(ID(str-rm-suffix(x, ":=(_)")), v) --> _.
```

```

  access-local(x, []) --> read-local(x).

  call-qualified(clos@ClosV(_, _, _, _, _, _, _, _, _, _, _), x, vs) --> call(clos, vs, x)
  where
    str-starts-with(x, "apply") == true.

  call-qualified(recv, x, vs) --> call(clos, vs, x)
  where
    lookup-local-method(recv, x) --> clos;
    log("disambiguate-closure, from qualified call") --> _;
    disambiguate-closure(clos, x) --> _.

  call(clos@ClosV(S, O, params, locals, code, L1, _, _, R, pts, rt), vs, name) :: L --> v :: L
  where
    log(name ++ " params:  " ++ str(params:AST) ++ "code: " ++ str(code:AST) ++ str(pts:AST))
  ↪  --> _;
    type-check(pts, vs) --> true;
    ensure-access(name, clos, S) --> _;
    add-locals(locals) :: L1 --> _ :: L2;
    update-locals(params, vs) :: L2 --> _ :: L3;
    S, O |- handle-return(code, R) :: L3 --> v :: L4.

  closure-source(ClosV(_, _, _, _, _, _, _, Src, _, _, _)) --> Src.

signature
  arrows
    do-return(V) --> V
    handle-return(Code, Return-Marker) --> V

rules

  R Return-To(r-mark) |- do-return(v) :: EX Ok() --> ??? :: EX Rex(r-mark, v).

  handle-return(code, No-Return()) --> v
  where
    code --> v.

  handle-return(code, R@Return-To(r-mark)) :: EX Ok() --> v :: EX
  where
    R |- code :: EX Ok() --> vcode :: EX1;
    case EX1 of {
      Rex(r-mark', vret) =>
        case eqI(r-mark', r-mark) of {
          true =>
            vret => v;
            Ok() => EX
          otherwise =>
            vcode => v;
            EX1 => EX
        }
      otherwise =>
        EX1 => EX;
        vcode => v
    }.

rules

  disambiguate-closure(clos, x) --> disambiguate-closure(clos, DoneV(), x).

  // closure was defined in bottom and found in local
  disambiguate-closure(clos@ClosV(_, _, _, _, _, _, _, src, _, _, _), _, _) --> clos
  where
    src-is-base(src) --> true.

  // closure was only found in local
  disambiguate-closure(clos@ClosV(_, _, _, _, _, _, _, _, _, _, _), DoneV(), _) --> clos.

  // closure was only found in outer
  disambiguate-closure(DoneV(), clos@ClosV(_, _, _, _, _, _, _, _, _, _, _), _) --> clos.
```

```
  // closure found in inherited and in outer
  disambiguate-closure(ClosV(_, _, _, _, _, _, _, src, _, _, _), ClosV(_, _, _, _, _, _, _,
  ↪  _, _, _ ,_), x) --> DoneV()
  where
    src-is-base(src) --> false;
    halt-error("Method '" ++ x ++ "' is defined both as an inherited/used" ++
      "field and in an enclosing scope.", "") --> _.

  // closure not found
  disambiguate-closure(DoneV(), DoneV(), x) --> DoneV()
  where
    halt-error("No such method: ", x) --> _.
```

trans/semantics/functions/functions.ds

```
module trans/semantics/functions/functions

imports
  src-gen/ds-signatures/grace-sig
  trans/semantics/values
  trans/semantics/statements
  trans/semantics/objectmodel
  trans/semantics/types
  trans/semantics/booleans
  trans/semantics/visibility
  trans/semantics/controlflow
  trans/semantics/functions/calls
  trans/semantics/expressions

signature
  arrows
    method-closure(Declaration) --> V
    block-closure(Declaration) --> V
    copy-closure(V) --> V

  native operators
    nativePrint: V -> V

rules


  MQCallOuter(e) --> outer(S)
  where
    e --> RefV(S).

  MCallL(ID(x), es) --> v
  where
    case x of {
      "print(_)" =>
        es => [e];
        nativePrint(e) => v
      "nativeIdentity(_)" =>
        es => [e];
        identity-check(e) --> v
      "while(_)do(_)" =>
        es => [e1, e2];
        while-loop(e1, e2) --> v

      otherwise =>
        call-implicit(x, es) --> v
    }.
```

```

  P Exec() |- MCallRecvL(e, ID(x), es) --> v
  where
    e --> recv;
    case recv of {
      BoolV(_) =>
        bool-call(recv, x, es) --> v
      NumV(_) =>
        num-call(recv, x, es) --> v
      StringV(_) =>
        str-call(recv, x, es) --> v
      TypeV(_) =>
        type-call(recv, x, es) --> v
      TypeV(_,_,_) =>
        type-call(recv, x, es) --> v
      otherwise =>
        call-qualified(recv, x, es) --> v
    }.

  BlockL(params, paramTypes, code) -->
    block-closure(MethodL(ID("lambda"), [], NoAnnotations(), params, paramTypes, no-type(),
  ↪  code)).

  Return(e) --> do-return(e).

rules /* closure construction */

  Src |- method-closure(MethodL(name, _, annos, params, paramTypes, returnType, code)) :: L
  ↪  -->
    clos :: L
  where
    current-self() --> S;
    current-outer() --> O;
    collect-locals(code, params) --> locals;
    error-check-locals(locals) --> _;
    ClosV(S, O, params, locals, code, L, has-anno-public(annos), Src, Return-To(fresh),
  ↪  paramTypes, returnType) => clos.

  block-closure(MethodL(_, _, _, params, paramTypes, _, code)) :: L -->
    ClosV(S, O, params, locals, code, L, true, src-base(), No-Return(), paramTypes,
  ↪  no-type()) :: L
  where
    current-self() --> S;
    current-outer() --> O;
    collect-locals(code, params) --> locals;
    error-check-locals(locals) --> _.

  copy-closure(ClosV(S, O, params, locals, code, L, _, Src, No-Return(), pt, rt)) -->
    ClosV(S, O, params, locals, code, L, false, Src', No-Return(), pt, rt)
  where
    Src |- src-previous() --> Src'.

  copy-closure(ClosV(S, O, params, locals, code, L, _, Src, Return-To(_), pt, rt)) -->
    ClosV(S, O, params, locals, code, L, false, Src', Return-To(fresh), pt, rt)
  where
    Src |- src-previous() --> Src'.
```

trans/semantics/functions/locals.ds

```
module trans/semantics/functions/locals

imports
  src-gen/ds-signatures/grace-sig
  trans/semantics/statements
  trans/semantics/functions/calls
```

```

signature
  sort aliases
    Env = Map(String, Addr)

  components
    L : Env

  arrows
    collect-locals(Code, List(Identifier)) --> List(Identifier)
    declaration-name(Declaration) --> Identifier


rules

  collect-locals([], xs) --> xs.

  collect-locals([c | code], xs) --> collect-locals(code, xs)
  where
    c =!=> Declaration(_).

  collect-locals([Declaration(d) | code], xs) --> collect-locals(code, [x | xs])
  where
    declaration-name(d) --> x.

  declaration-name(VariableL(x, _, _, _)) --> x.

  declaration-name(ConstantL(x, _, _, _)) --> x.


/* ======== local variable error checking ========= */
signature
  arrows
    error-check-locals(List(Identifier)) --> U

    ensure-valid-local(String) --> U

rules

  error-check-locals([]) --> U().

  error-check-locals([WildCard() | ids]) --> error-check-locals(ids).

  error-check-locals([ID(x) | ids]) :: L --> U() :: L
  where
    ensure-valid-local(x) --> _;
    error-check-locals(ids) :: L { x |--> 0, L} --> _.

  ensure-valid-local(x) --> U()
  where
    is-local(x) --> true;
    halt-error("Local '" ++ x ++ "' may not redefine local method.", "") --> _.

  ensure-valid-local(x) --> U()
  where
    is-local(x) --> false;
    lookup-local-method(current-self(), x) --> ClosV(_, _, _, _, _, _, _, _, _, _, _);
    halt-error("Local '" ++ x ++ "' may not redefine method from self.", "") --> _.

  ensure-valid-local(x) --> U()
  where
    is-local(x) --> false;
    lookup-outer-method(current-outer(), x) --> ClosV(_, _, _, _, _, _, _, _, _, _, _);
    halt-error("Local '" ++ x ++ "' may not shadow method from an enclosing scope.", "") -->
  ↪  _.

  ensure-valid-local(_) --> U().


/* ======== local environment operations ========== */
```

```
signature
  arrows
    add-local(Identifier) --> U
    add-locals(List(Identifier)) --> U

    update-local(Identifier, V) --> U
    update-locals(List(Identifier), List(V)) --> U

    is-local(String) --> Bool
    read-local(String) --> V
    snapshot-locals() --> Env

rules

  add-local(ID(x)) :: L --> U() :: L {x |--> addr, L}
  where
    v-allocate(UninitializedV()) --> addr.

  add-local(WildCard()) --> U().

  add-locals([]) --> U().

  add-locals([x | xs]) --> add-locals(xs)
  where
    add-local(x) --> _.

  update-locals([], []) --> U().

  update-locals([id | ids], [v | vs]) --> update-locals(ids, vs)
  where
    update-local(id, v) --> _.

  update-local(ID(x), v) :: L --> U() :: L
  where
    v-update(L[x], v) --> _.

  update-local(WildCard(), _) --> U().

  is-local(x) :: L --> is-local :: L
  where
    ":=(_)" => bind_suffix;
    str-ends-with(x, bind_suffix) --> is-assign;
    case is-assign of {
      true =>
        str-rm-suffix(x, bind_suffix) --> x';
        L[x'?] => is-local
      false =>
        L[x?] => is-local
    }.

  read-local(x) :: L --> ensure-defined(v-read(L[x])) :: L.

  snapshot-locals() :: L --> L :: L.


/* ============ variable heap operations ======== */

signature
  sort aliases
    Addr = Int
    VHeap = Map(Addr, V)

  components
    VH : VHeap

  arrows
    v-allocate(V) :: H --> Addr :: H
    v-update(Addr, V) :: H --> Addr :: H
    v-read(Addr) :: H --> V :: H
    v-next() --> Addr
```

```

rules

  v-allocate(v) :: VH --> addr :: VH {addr |--> v, VH}
  where
    v-next() --> addr.

  v-read(addr) :: VH --> VH[addr] :: VH.

  v-update(addr, v) :: VH --> addr :: VH {addr |--> v, VH}.

  v-next() --> fresh.
```

trans/semantics/statements.ds

```
module trans/semantics/statements

imports
  src-gen/ds-signatures/grace-sig
  trans/semantics/expressions
  trans/semantics/values

signature
  sorts V
  constructors
    DoneV : V

  sort aliases
    Code = List(Statement)

  arrows
    Statement --> V
    Code --> V

rules

  // unwrap expression
  Expression(e) --> e.

  Dialect(_) --> DoneV().

  Declaration(VariableL(x, _, _, e)) --> v
  where
    e --> v;
    case x of {
      ID(_) =>
        update-local(x, v) --> _
      WildCard() =>
    }.

  Declaration(ConstantL(x, _, _, e)) --> v
  where
    e --> v;
    case x of {
      ID(_) =>
        update-local(x, v) --> _
      WildCard() =>
    }.

```

trans/semantics/store.ds

```
module trans/semantics/store

signature
  sorts
    HeapData

  sort aliases
    Addr = Int
    H = Map(Addr, HeapData)

  components
    H : H

  arrows
    allocate(HeapData) :: H --> Addr :: H
    update(Addr, HeapData) :: H --> Addr :: H
    is-stored(Addr) :: H --> Bool :: H
    read(Addr) :: H --> HeapData :: H
    next() --> Addr

rules

  allocate(data) :: H --> addr :: H {addr |--> data, H}
  where
    next() --> addr.

  is-stored(addr) :: H --> H[addr?] :: H.

  read(addr) :: H --> H[addr] :: H.

  update(addr, data) :: H --> addr :: H {addr |--> data, H}.

  next() --> fresh.

```

trans/semantics/types.ds

```
module trans/semantics/types

imports
  src-gen/ds-signatures/grace-sig
  trans/semantics/visibility
  trans/semantics/store
  trans/semantics/objectmodel

signature

  sorts
    TypeOp

  sort aliases
    Type = List(TypeRule)

  constructors
    TypeV: List(TypeRule) -> V
    TypeV: TypeOp * V * V -> V
    UnkwnV: V
    Variant: TypeOp
    Intersection: TypeOp
    Subtraction: TypeOp
    Union: TypeOp

  arrows
```

```
    TypeExp --> V
    List(TypeExp) --> List(V)

    type-check(List(V), List(V)) --> Bool

    no-type() --> TypeExp
    new-type(List(TypeRule)) --> V
    type-call(V, String, List(Exp)) --> V
    get-type(V) --> V
    get-names(V) --> List(String)
    get-object-names(Addr) --> List(String)
    get-object-type(Addr) --> V
    get-type-methods(Type) --> List(String)
    methods-to-list(Methods) --> List(String)
    methods-to-type(Methods) --> Type

    names-to-type(List(String)) --> Type

    compare-types(V, V) --> Bool

    compare-names(List(String), List(String)) --> Bool

    contains-name(String, List(String)) --> Bool

rules

  AnonType(TypeBlock(trs)) --> new-type(trs).

  [] : List(TypeExp) --> [] : List(V).
  [te | tes] : List(TypeExp) --> [tv | tvs] : List(V)
  where
    te --> tv;
    tes --> tvs.

  TypeExp(t) --> t.

  Variant(v1, v2) --> TypeV(Variant(), v1, v2).

  Unkwn() --> UnkwnV().

  TypeID(ID(name), _) --> type
  where
    call-implicit(name, []) --> type.

  no-type() --> Unkwn().

  type-check([] , []) --> true.
  type-check([pt | pts] , [v | vs]) --> type-check(pts, vs)
  where
    compare-types(pt, v) --> true.

  type-check(ptypes@[pt | pts] , vtypes@[v | vs]) --> type-check(pts, vs)
  where
    log("parameter types:" ++ str(ptypes:AST) ++ " value types:" ++ str(vtypes:AST)) --> _;
    compare-types(pt, v) --> false;
    halt-error("Type mismatch!", "") --> _.


  new-type(trs) --> TypeV(trs).

  type-call(t, "match(_)", [other]) --> BoolV(compare-types(t, get-type(other))).

  get-type(v) --> v
  where
    "getting type of: " => prefix;
    case v of {
      RefV(addr) =>
        get-object-type(addr) --> v;
        "Object Ref" => type
      StringV(_) =>
```

```
        TypeV([]) => v;
        "String V" => type
      BoolV(_) =>
        TypeV([]) => v;
        "Boolean V" => type
      tv@TypeV(_) =>
        tv => v;
        "Type V" => type
      tv@TypeV(_,_,_) =>
        tv => v;
        "Type V expr" => type
      otherwise =>
        TypeV([]) => v;
        "Unknown V" => type
    };
    log(prefix ++ type) --> _.

  get-names(v) --> t
  where
    case v of {
      RefV(addr) =>
        get-object-names(addr) --> t
      NumV(_) =>
        [] => t
      StringV(_) =>
        [] => t
      BoolV(_) =>
        [] => t
      TypeV(tr) =>
        get-type-methods(tr) --> t
      UninitializedV() =>
        [] => t
      otherwise =>
        [] => t;
        halt-error("Unknown V to get type for: ", str(v)) --> _
    }.

  get-object-type(addr) --> TypeV(t)
  where
    read(addr) --> Obj(_,_,_,methods);
    methods-to-type(methods) --> t.

  get-object-names(addr) --> ls
  where
    read(addr) --> Obj(_,_,_,methods);
    methods-to-list(methods) --> ls.

  get-type-methods([]) --> [] : List(String).
  get-type-methods([TypeRuleL(ID(n), _, _, _) | trs]) --> [n | get-type-methods(trs)].

  methods-to-type(methods) --> trs
  where
    methods-to-list(methods) --> mls;
    names-to-type(mls) --> trs.

  names-to-type([]) --> [].
  names-to-type([s|ss]) --> [TypeRuleL(ID(s), [], [], no-type()) | names-to-type(ss)].

  methods-to-list(map) --> methodnames
  where
    allkeys(map) => methodnames.

  compare-types(UnkwnV(), _) --> true.

  compare-types(TypeV(Variant(), t1, t2), t3@TypeV(_)) --> b
  where
    compare-types(t1, t3) --> res;
    case res of {
      true =>
        true => b
```

```
      otherwise =>
        compare-types(t2, t3) --> b
    }.

  compare-types(_, UninitializedV()) --> true.

  compare-types(t1@TypeV(_), v) -->
    compare-names(get-names(t1), get-names(v))
  where
    log("comparing two type sigs, 1: " ++ str(t1) ++ ", 2: " ++ str(v)) --> _.

  compare-names([], _) --> true.

  compare-names(t1@[_|_],[]) --> false
  where
    log("type doesn't conform because t2 doesn't contain the types: " ++ str(t1:AST)) --> _.

  compare-names([t1|t1s], t2s) --> compare-names(t1s, t2s)
  where
    contains-name(t1, t2s) --> true.

  compare-names([t1|_], t2s) --> false
  where
    contains-name(t1, t2s) --> false;
    log("comparing names, t2 is missing a type that t1 has") --> _.

  contains-name(_, []) --> false.

  contains-name(s, [s' | _]) --> true
  where
    s == s'.

  contains-name(s, [s' | ss]) --> contains-name(s, ss)
  where
    s != s'.
```

trans/semantics/values.ds

```
module trans/semantics/values

imports
  trans/semantics/runtime/natives

signature

  sorts
    V
    U

  constructors
    UninitializedV: V
    U : U

  variables
    v : V
    vs : List(V)

  arrows
    ensure-defined(V) --> V

rules
  ensure-defined(v) --> v
  where
    v =!=> UninitializedV().

```

```
  ensure-defined(v@UninitializedV()) --> v
  where
    halt-error("Read of an uninitialised value attempted", "") --> _.
```

trans/semantics/visibility.ds

```
module trans/semantics/visibility

imports
  src-gen/ds-signatures/grace-sig
  src-gen/ds-signatures/grace-lowered-sig

imports
  trans/semantics/values
  trans/semantics/store
  trans/semantics/runtime/natives
  trans/semantics/strings
  trans/semantics/numbers
  trans/semantics/lineups
  trans/semantics/objectmodel
  trans/semantics/statements
  trans/semantics/imports


/* ======= REACHABILITY CHECK ======= */
signature
  sort aliases
    HeapData = Object
  arrows
    ensure-access(String, V, Addr) --> V

    can-reach(Addr) --> Bool
    can-reach-map(List(Addr), Addr) --> Bool

rules

  ensure-access(x, clos@ClosV(_, _, _, _, _, _, true, _, _, _, _), recv) --> clos.

  ensure-access(x, clos@ClosV(_, _, _, _, _, _, false, _, _, _, _), recv) --> clos
  where
    can-reach(recv) --> visible;
    case visible of {
      false =>
            halt-error("Requested confidential method '" ++ x ++ "' of object: " ++
↪  int2string(recv) ++ " from outside", "") --> _
      otherwise =>
    }.

  S |- can-reach(S') --> true
  where
    S == S'.

  S |- can-reach(S') --> false
  where
    S != S';
    is-stored(S) --> false.

  S |- can-reach(S') --> maybe
  where
    S != S';
    is-stored(S) --> true;
    read(S) --> Obj(_, outers, _, _);
    can-reach-map(outers, S') --> maybe.

  can-reach-map([], _) --> false.
```

```
  can-reach-map([S | ss], S') --> reachable'
  where
    S |- can-reach(S') --> reachable;
    case reachable of {
      false =>
        can-reach-map(ss, S') --> reachable'
      otherwise =>
        true => reachable'
    }.



/* ==== VISIBILITY ANNOTATION PROCESSING ==== */
signature

  arrows
    has-anno-readable(Annotations) --> Bool
    has-anno-writable(Annotations) --> Bool
    has-anno-confidential(Annotations) --> Bool
    has-anno-public(Annotations) --> Bool

    has-anno(List(Annotation), Annotation) --> Bool

    visibility-annos(Bool) --> Annotations

rules

  has-anno-readable(Annotations(annos)) --> has-anno(annos, Readable()).

  has-anno-writable(Annotations(annos)) --> has-anno(annos, Writable()).

  has-anno-confidential(Annotations(annos)) --> has-anno(annos, Confidential()).

  has-anno-public(Annotations(annos)) --> has-anno(annos, Public()).

  has-anno([], _) --> false.

  has-anno([anno | _], anno') --> true
  where
    anno == anno'.

  has-anno([anno | annos], anno') --> has-anno(annos, anno')
  where
    anno != anno'.

  visibility-annos(true) --> Annotations([Public()]).

  visibility-annos(false) --> Annotations([Confidential()]).
```

trans/semantics/booleans.ds

```
module trans/semantics/booleans

imports
  src-gen/ds-signatures/grace-lowered-sig
  trans/semantics/expressions
  trans/semantics/values

signature
  constructors
    BoolV : Bool -> V

  arrows
    bool-call(V, String, List(Exp)) --> V
```

```

  native operators
    bool-call-native: String * V * V -> V
    bool-call-native: String * V -> V

rules

  Boolean(True()) --> BoolV(true).

  Boolean(False()) --> BoolV(false).

  bool-call(v, x, []) --> bool-call-native(x, v).

  bool-call(v1, x, [v2@BoolV(_)]) --> bool-call-native(x, v1, v2).

  bool-call(BoolV(true), "ifTrue(_)ifFalse(_)", [e1, _]) --> call(e1, [], "apply").

  bool-call(BoolV(false), "ifTrue(_)ifFalse(_)", [_, e2]) --> call(e2, [], "apply").


  bool-call(BoolV(true), "ifTrue(_)", [e]) --> call(e, [], "apply").

  bool-call(BoolV(false), "ifTrue(_)", [_]) --> DoneV().


  bool-call(BoolV(true), "ifFalse(_)", [_]) --> DoneV().

  bool-call(BoolV(false), "ifFalse(_)", [e]) --> call(e, [], "apply").

  bool-call(BoolV(true), "asString", []) --> StringV("true").
  bool-call(BoolV(false), "asString", []) --> StringV("false").
```

trans/semantics/strings.ds

```
module trans/semantics/strings

imports
  src-gen/ds-signatures/grace-lowered-sig
  trans/semantics/expressions
  trans/semantics/values
  trans/semantics/runtime/natives

signature
  constructors
    StringV : String -> V

  arrows
    str-call(V, String, List(Exp)) --> V
    str-call-evaluated(String, V, V) --> V

  native operators
    string-call-native: String * V * V -> V
    string-call-native: String * V -> V

rules

  String(s) --> StringV(s).

  str-call(v1, op, [v2]) --> str-call-evaluated(op, v1, v2).

  str-call(v, op, []) --> string-call-native(op, v).

  str-call-evaluated(op, v1, v2@StringV(_)) --> string-call-native(op, v1, v2).

  str-call-evaluated(op, v1, NumV(i)) --> string-call-native(op, v1, StringV(s))
```

```
  where
    int2str(i) --> s.

  str-call-evaluated(”==(_)”, _, v2) --> BoolV(false)
  where
    v2 =!=> StringV(_).
```

trans/semantics/numbers.ds

```
module trans/semantics/numbers

imports
  src-gen/ds-signatures/grace-lowered-sig
  trans/semantics/expressions
  trans/semantics/values
  trans/semantics/runtime/natives
  trans/semantics/booleans

signature
  constructors
    NumV : Int -> V

  arrows
    num-call(V, String, List(Exp)) --> V
    num-call-evaluated(String, V, V) --> V

  native operators
    num-call-native: String * V * V -> V
    num-call-native: String * V -> V

rules

  Number(a) --> NumV(string2int(a)).

  num-call(v1, x, [v2]) --> num-call-evaluated(x, v1, v2).

  num-call(v1, x, []) --> num-call-native(x, v1).

  num-call-evaluated(”==(_)”, _, v2) --> BoolV(false)
  where
    v2 =!=> NumV(_).

  num-call-evaluated(”++(_)”, NumV(i1), StringV(s2)) --> StringV(s1 ++ s2)
  where
    int2str(i1) --> s1.

  num-call-evaluated(x, v1, v2) --> num-call-native(x, v1, v2).
```

trans/semantics/lineups.ds

```
module trans/semantics/lineups

imports
  src-gen/ds-signatures/grace-sig
  trans/semantics/expressions

signature
  constructors
    LineupV : List(V) -> V

```

```
rules

  LineupExp(Lineup(vs)) --> LineupV(vs).
```

trans/semantics/runtime/natives.ds

```
module trans/semantics/runtime/natives

imports
  trans/semantics/values

signature
  native operators
    parseI : String -> Int
    error: String * String -> String
    addI: Int * Int -> Int
    int2string: Int -> String
    str: AST -> String
    eqI: Int * Int -> Bool
    gtI: Int * Int -> Bool
  arrows
    string2int(String) --> Int
    int2str(Int) --> String
    halt-error(String, String) --> String

rules

  string2int(s) --> parseI(s).
  int2str(i) --> int2string(i).

  halt-error(s1, s2) --> error(s1, s2).


/* string ops */

signature
  native operators
    logdebug: String -> String
    str_starts_with : String * String -> Bool
    str_ends_with : String * String -> Bool
    str_remove_suffix : String * String -> String

  arrows
    concat(List(String)) --> String
    separate-by(List(String), String) --> List(String)
    log(String) --> String
    str-starts-with(String, String) --> Bool
    str-ends-with(String, String) --> Bool
    str-rm-suffix(String, String) --> String

rules

  concat([]) --> "".

  concat([s | ss]) --> s ++ ss'
  where
    concat(ss) --> ss'.

  separate-by([], _) --> [].

  separate-by([s], _) --> [s].

  separate-by([s1| xs@[_ | _]], sep) --> [s1, sep | xs']
  where
    separate-by(xs, sep) --> xs'.
```

```
str-starts-with(s, prefix) --> str_starts_with(s, prefix).

str-ends-with(s, suffix) --> str_ends_with(s, suffix).

str-rm-suffix(s, suffix) --> str_remove_suffix(s, suffix).

log(s) --> s
where
  logdebug(s) => _.
```