INSTITUTE FOR INFORMATICS
Ludwig Maximilian University of Munich

Conception and Implementation of an Object-Oriented Mini-Language

Daniel Maier
(matriculation number: 11373750)

Bachelor thesis

Supervisor	Prof. Dr. François Bry
Submission date	18/08/2022

Declaration of Authenticity

I herewith declare that I wrote this thesis on my own and did not use any other sources or any other help than those mentioned.

Munich, the 18/08/2022 Daniel Maier

Abstract

This thesis introduces an imperative, statically typed and object-oriented mini-language. Along with the language introduction, the core features of object-oriented programming are highlighted. For this, various program examples are presented. An implementation of the language using the Haskell programming language is also provided. This implementation serves two purposes. Firstly, the reader is encouraged to run the example programs and also run their own programs using the implementation. Secondly, the latter part of the thesis describes the implementation in detail. It uses the established combination of parsing and code generation to translate the program to machine code. The target machine of the code generator is a simulated abstract machine that is also part of the Haskell implementation.

Zusammenfassung

Diese Bachelorarbeit führt eine imperative, statisch getypte und objektorientierte Mini-Sprache ein. Zusammen mit der Einführung der Sprache wird die Kernfunktionalität von objektorientierter Programmierung hervorgehoben. Hierfür werden verschiedene Programmbeispiele präsentiert. Eine Implementierung der Sprache, die die Programmiersprache Haskell verwendet, wird ebenfalls bereitgestellt. Diese Implementierung erfüllt zwei Zwecke. Erstens wird der Leser ermutigt, die Beispielprogramme, und auch selbst geschriebene Programme, unter Verwendung der Implementierung auszuführen. Zweitens wird im hinteren Teil der Arbeit die Implementierung im Detail beschrieben. Diese verwendet die etablierte Kombination aus Parsing und Code-Erzeugung, um das Programm in Maschinencode zu übersetzen. Die Zielmaschine des Code-Erzeugers ist eine simulierte abstrakte Maschine, die ebenfalls Teil der Haskell-Implementierung ist.

Acknowledgments

First, I want to thank Prof. Dr. François Bry for his ongoing support, especially when the task seemed overwhelming. Without his unique approach to teaching about programming languages, getting into the subject would have seemed insurmountable.

I also want to thank my fellow students Sven-Gerrit Kluge and Sabbir Ahmed whom I collaborated with on a prior project on programming languages. Some of the ideas for the implementation discussed in this thesis originated from that project and were developed with them.

Contents

CHAPTER 1

Introduction

The history of object-oriented programming (OOP) dates back to the 1970s with languages like Smalltalk, Java, C++ and many others. Today, OOP is a core feature of some of the most widely used programming languages. Yet, there is much confusion about what exactly constitutes the essence of OOP. This is at least partly due to the fact that almost all modern programming languages that offer support for OOP do so in different ways and also offer many additional features that are not essential to OOP in and of itself.

This thesis introduces an imperative, statically typed and object-oriented mini-language that is limited to the most important features of OOP. The language is not intended to serve as an example of a fully-featured production-grade language, but rather as a didactic means to better understand the involved concepts and their implementation. For the provided implementation (see [10]), the Haskell programming language is used. Although Haskell might be regarded as a niche language, the author intends to show that it is particularly suitable for describing the implementation of programming languages in a clear and concise way.

Chapter 2 defines the most essential aspects of object-oriented programming relevant today. Based on this definition, the design of the aforementioned language is described in chapter 3. After briefly introducing the compiler concepts used for the implementation in chapter 4, the implementation itself is discussed in chapter 5 to chapter 7. Mentioning perspectives for future work, the thesis is concluded in chapter 8.

CHAPTER 2

Concepts of Object-Oriented Programming Languages

Object-oriented programming is a collection of interlinked concepts that has organically evolved over the last 50 years. This has lead to a general sense of ambiguity around OOP and a dissent over specific implementation details. Here, the author tries to lay out a useful definition of the important concepts to build the language upon.

Today, there is an authoritative definition of object-oriented concepts, which is from the ISO standard for information technology vocabulary (see [7]). It gives a concise definition of what *object-oriented* means:

Definition 1: ***object-oriented***

pertaining to a technique or a programming language that supports objects, classes, and inheritance

So a (minimal) object-oriented programming language needs to support objects, classes and inheritance. But what are *objects*, *classes* and *inheritance*? Fortunately, the standard defines these terms as well:

Definition 2: ***object***

set of operations and data that store and retain the effect of the operations

This means that objects differ from traditional (mutable) *structures* in languages like C in that they carry an additional set of operations with their data. They differ from mathematical objects in that in general, they are not automatically equal if they contain the same data and operations, which is why they are best understood carrying an additional *unique identifier* with them that clearly distinguishes them from other objects. Even though it is often debated that this is an implementation detail which should not be raised into the context of a fundamental definition, an understanding of this notion of objects is incomplete without it. This definition is consistent with the way objects are defined in most popular object-oriented languages like Java and C++.

The notion of a *class* builds on objects:

Definition 3: ***class***

template for objects that defines the internal structure and the set of operations for instances of such objects

In other words, a class is a definition of the data structures and operations for any object that is an instance of that class.

Finally, *inheritance*:

Definition 4: ***inheritance***

copying of all or part of the internal structure and of the set of operations from one class to a subordinate class

Instead of this procedural notion of inheritance that hints explicitly at a possible implementation of the concept, one can equivalently understand it as a transitive relation between classes, where the *subordinate* class or *sub*class inherits the data structures and operations of the *upper* class, just by declaring its subordination, without declaring the inherited structures explicitly.

CHAPTER 3

The Mini-Language O

This chapter introduces step by step the syntax and semantics of the language O, and motivates the relevant decisions made in the design process. Throughout the chapter, different example programs are provided for illustration of certain concepts or problems. All example programs can directly be run from the provided implementation's folder `resources/example-programs` (see [10]) without having to copy them by hand from this thesis.

3.1 Lexical Syntax

The *Lexical Syntax* of a language decides how the character sequence describing the program is split up into *lexemes* or *tokens*. Underlying this is a formal grammar that is regular for most common programming languages - this is also true for O. Without defining this lexical grammar for the language O explicitly, it suffices to say that a lexeme for O can be one of the following:

- a *symbol name* that begins with a small letter and consists of only small and capital letters (commonly called *lowerCamelCase*),
- a *class name* that begins with a capital letter and consists of only small and capital letters (commonly called *UpperCamelCase*),
- a *string* that begins and ends with the character `'"'`, and otherwise contains arbitrary other characters,
- an *integer* that consists of only digits,
- one of `USING`, `CLASS`, `SUBCLASSOF`, `FIELDS`, `INIT`, `INT`, `OBJ`, `PROCEDURE`, `METHOD`, `RETURNS`, `CALL`, `READ`, `IF`, `THEN`, `WHILE`, `DO`, `PRINTI`, `PRINTS`, `PRINTLNS`, `ERROR`, `NOT`, `:=`
- or one of the characters `= , . > < + - * / ( ) [ ] { }`

Any two consecutive lexemes must be separated by at least one white space, end of line or tab character.

The process of *lexical analysis*, that is checking for lexical errors and producing the corresponding lexemes, is carried out by the *tokenizer* described in chapter 5.

3.2 Simple programs and instruction overview

As mentioned in chapter 1, the language is *imperative*. This means that instructions are used to tell the machine explicitly which actions to perform in order to carry out the computation. An O-program is a sequence of such instructions, along with an optional set of classes and procedures, which are introduced at a later point. In its simplest form, an O-program can be very short, as hello-world program 3.1 shows.

```
DO { PRINTLNS "Hello world!" }
```

Listing 3.1: Example program `hello.olang`

Program 3.1 starts with `DO` to indicate the start of the *main program*, which is the sequence of instructions to execute. `PRINTLNS` is the instruction that prints out the supplied string and concludes with a new line. As expected, the program produces the following output:

```
Hello World!
```

O provides instructions and expressions similar to those defined for the mini-language I in [3]. The basic instructions are described in Table 3.1. Instead of a complete specification of the functionality, it should rather be regarded as an overview. Most primitives are discussed later in greater detail.

Instruction	**Meaning**
`n := expr`	Assigns the value of expression `expr` to variable `n`
`INT n`	Declares a new integer variable with name `n`
`OBJ T n`	Declares a new object variable of type `T` with name `n`
`CALL proc(...)`	Invokes procedure `proc` with parameters `(...)`
`READ n`	Reads an integer value from the standard input and assigns the value to `n`
`PRINTI expr`	Prints value of expression `expr` to the standard output
`PRINTS str`	Prints string `str` to the standard output
`PRINTLNS str`	Prints string `str` followed by a new line character to the standard output
`ERROR`	Terminates the program (an error is encountered)

Table 3.1: The basic O-instructions

Expressions are either basic or composite. Basic expressions are integer values, variable references, field references, or procedure calls, method calls or class instantiations with empty parameter lists. Composite expressions are either arithmetic expressions combining expressions with the operators `+ - * /`, or procedure calls, method calls or class instantiations with non-empty parameter lists. Round brackets can be used in arithmetic expressions to indicate the order of evaluation - if they are not used, evaluation follows the usual order of operations. The division `/` is implemented by *integer division*, which always yields an integer value by truncating the result if necessary.

Note that a variable declaration will shadow any variable of the same name declared earlier, like in many imperative languages. This should be avoided in general, but can be useful in conjunction with the *scoping* mechanism introduced later. Integer variable declarations will assign a default value of 0, whereas object variable declarations will assign an invalid address.

Most languages support the declaration of boolean variables, fractional number variables and string variables. To keep the syntax and implementation simple, O omits these other types of

variables. Unfortunately, the omission of boolean variables results in the necessity to utilize an arguably bad coding style of using integer variables to store truth values, where 0 represents a truth value of false, and a non-zero integer value represents a truth value of true. It appears to the author that this is a trade-off worth making.

Using only basic instructions, the very simple calculator program 3.2 for adding two integers can be created.

```
DO {
    PRINTLNS "This program calculates the sum of two integers a + b."
    INT a
    INT b
    PRINTS "Please enter a: "
    READ a
    PRINTS "Please enter b: "
    READ b
    PRINTS "a + b = "
    PRINTI a + b
}
```

Listing 3.2: Example program `sum.olang`

Running program 3.2 with the integers 1 and 2 yields:

```
This program calculates the sum of two integers a + b.
Please enter a: 1
Please enter b: 2
a + b = 3
```

In addition to the basic instructions, the composite instructions described in Table 3.2 are provided.

Instruction	**Meaning**
`{ seq }`	Executes the instructions in `seq` in sequence
`IF cond THEN cmd`	If condition `cond` holds, then executes instruction (or sequence of instructions) `cmd`
`WHILE cond DO cmd`	As long as condition `cond` holds, repeatedly executes instruction (or sequence of instructions) `cmd`

Table 3.2: The composite O-instructions

Conditions are either a comparison of two expressions using one of the comparators `< = >`, or a negation of another condition using the keyword `NOT`.

Note that `{...}` introduces a new *scope*, meaning the variables declared within are only visible within - any reference to an inner variable from outside the scope will lead to a compile-time error.

This can already be used to implement considerably more complicated algorithms, like a primitive prime sieve:

```
DO {
    PRINTLNS "I will now begin listing all primes"
    INT n
    n := 2
    WHILE n > 0 DO {
```

```
        INT m
        INT isprime
        m := 2
        isprime := 1
        WHILE m < n DO {
                IF (n / m) * m = n THEN isprime := 0
                m := m + 1
        }
        IF NOT isprime = 0 THEN {
            PRINTI n
            PRINTLNS ""
        }
        n := n + 1
    }
}
```

Listing 3.3: Example program `primes.olang`

Program 3.3 will eventually list all primes and consequently never halts:

```
I will now begin listing all primes
2
3
5
7
11
13
...
```

Note that program 3.3 is only capable of listing any prime if an integer variable may assume any integer value. In contrast to most programming languages, where an integer value has a fixed lower and upper bound somewhere between -2^{64} and $+2^{64}$, the integer values in O have no such strict lower and upper bound. Instead, the lower and upper bound depend on the machine and operating system that the program is running on. The bound is reached if there is not enough memory for the program to represent the value. Generally, this means that an integer value in O can become *very* large, at the cost of making arithmetic operations slower.

3.3 Procedures

Writing a program as a sequence of instructions can become laborious quickly, especially once the need for re-using a certain part of the program as a subroutine arises. To allow for this, O provides *procedures*. A procedure is, like the classes that are introduced later, defined in the preamble of a program like follows:

```
USING [
    PROCEDURE foo(INT a, INT b) RETURNS INT c {
        ... procedure code ...
    }
] DO {
    ... main program code ...
}
```

This defines a procedure `foo` with two integer parameters `a` and `b`, returning an integer `c`.

There can be any number of formal parameters of arbitrary type. For now, it suffices to say that a formal parameter `n` can be an integer parameter (`INT n`) or an object parameter of some type `T` (`OBJ T n`) - types are discussed in detail at a later point in the chapter, and again in chapter 7. Return parameters are optional - there can be zero or one return parameter defined. Multiple return parameters are not allowed. Once defined, procedures can be invoked (or *called*), which involves providing argument expressions of the correct type. At run-time, the values of the provided argument expressions will be copied, and the copies are bound to the corresponding names declared in the formal parameter list of the procedure. This behaviour is commonly referred to as *call by value*. If a return parameter is defined, it will be treated as a declaration, which effects to initializing it with the default value. Then, the procedure code is executed. When the procedure finishes execution, it yields the value of its return parameter at that point. If a procedure has a return parameter, it is invoked as part of an expression. If it does not, it is invoked using the `CALL` instruction. This distinction always makes clear when a return value is expected. A procedure can be invoked in the main program, invoke itself, or be invoked in another procedure defined later. In principle, the language could also support indirect recursion, but due to a shortcoming in the code generator's implementation (see chapter 7), the provided implementation does not allow for that.

Additionally, a procedure may have a preamble to its code which can define a set of sub-procedures for it to use:

```
PROCEDURE foo(INT a, INT b) RETURNS INT c
USING [
    PROCEDURE bar(INT a) {
        ... bar procedure code ...
    }
]
{
    ... foo procedure code ...
}
```

These sub-procedures can only be invoked from the procedure they are defined within.

Note that a procedure only has access to its arguments and the variables it declares in its code, so the input it relies on to carry out the computation must be supplied either through the arguments, or the usage of the `READ`-instruction in the procedure's code (or the indirect usage thereof in another procedure that is invoked). This makes procedures modular to a high degree, but in general they are still subject to side effects and therefore *referentially opaque*.

Procedures are not only a way for outsourcing code - together with recursion they provide a powerful way of expressing computations. Consider the ackermann function, $ack : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, as defined by Robinson (see [12]):

$$\begin{aligned} ack(0,m) &= m+1 \\ ack(n+1,0) &= ack(n,1) \\ ack(n+1,m+1) &= ack(n, ack(n+1,m)) \end{aligned}$$

Using recursive procedures, program 3.4 for calculating the ackermann function can be created:

```
USING [
  PROCEDURE ack(INT n, INT m) RETURNS INT a {
    IF n < 0 THEN {
      PRINTLNS "ERROR: n is not a natural number!"
      ERROR
    }
```

```
    IF m < 0 THEN {
      PRINTLNS "ERROR: m is not a natural number!"
      ERROR
    }
    IF NOT n < 0 THEN
      IF NOT m < 0 THEN {
        IF n = 0 THEN a := m + 1
        IF NOT n = 0 THEN {
          IF m = 0 THEN a := ack(n - 1, 1)
          IF NOT m = 0 THEN a := ack(n - 1, ack(n, m - 1))
        }
      }
  }
] DO {
  PRINTLNS "This program calculates the ackermann function ack(n, m)."
  INT n
  INT m
  PRINTS "Please enter a natural number n: "
  READ n
  PRINTS "Please enter a natural number m: "
  READ m
  PRINTS "ack(n, m) = "
  PRINTI ack(n, m)
}
```

Listing 3.4: Example program `ackermann.olang`

Running program 3.4 with $n = 3$ and $m = 6$ yields:

```
This program calculates the ackermann function ack(n, m).
Please enter a natural number n: 3
Please enter a natural number m: 6
ack(n, m) = 509
```

The fact that the ackermann function is not a primitive recursive function (see [12]) provides evidence for the computational power of O. More discussion on this topic, including a more elaborate program example involving the dynamic data structure of linked lists, can be found in chapter 9.

3.4 Objects and classes

As established in chapter 2, an object is an identifiable collection of mutable data along with operations on this data, and a class is a template for such objects. Since in O objects can only be created from class templates (as class *instances*), it makes sense to talk about classes first.

In O, a simple case of a class without methods and only one field, in the preamble of some program, looks like the following:

```
USING [
    ...
    CLASS Intbox(INT i)
    FIELDS INT i
    INIT { this.i := i }
    ...
```

```
7 ] DO {
8     ...
9 }
```

Here, `CLASS Intbox` declares a class with name `Intbox`, and `FIELDS INT i` translates to any instance of the class having a data field named `i` that holds an integer value. `INIT { this.i := i }` defines the class's *initializer* that is some code which is executed upon object creation (class initialization). In this case, the initializer code is just one instruction that assigns the initializer parameter (`INT i` on line 1) to the new object's field that is also named `i`. To refer to the object that is being created, the identifier `this` is used.

Declaring a class in the program's preamble also introduces a type of the same name that can be used in declarations. Declaring an object variable named `o` of type `Intbox` is simple:

```
OBJ Intbox o
```

Note that this declaration **does not** create an object of class `Intbox`! It merely creates a container for an identifier (an *address*) of objects of class `Intbox` (commonly referred to as a *pointer*). In comparison, the declaration

```
INT i
```

creates a container for an integer value, which is also initialized with a value of 0.

Creating an object of a class is done by invoking the class's initializer like a procedure with the class's name, providing all the necessary arguments. In case of class `Intbox`, an object can be created with the expression `Intbox(1)`. By checking the initializer code, it can be verified that the field `i` of this newly created object has the integer value `1`.

Object fields can be accessed by dot notation on object variables. A field `i` of an object referred to by the object variable `a` is accessed by the field reference `a.i`. So verifying the above claim about the value of field `i` can be accomplished by running a test program:

```
USING [
    CLASS Intbox(INT i)
    FIELDS INT i
    INIT { this.i := i }
] DO {
    OBJ Intbox i
    i := Intbox(1)
    PRINTI o.i
}
```

Listing 3.5: Example program `intbox0.olang`

Running program 3.5 yields the expected value of 1.

The level of indirection introduced by object variables has consequences that might not be immediately obvious. To illustrate this, consider the following example with two procedures:

```
USING [
    CLASS Intbox(INT i)
    FIELDS INT i
    INIT { this.i := i }

    PROCEDURE setZero(OBJ Intbox b) {
        b.i := 0
    }
```

```
    PROCEDURE setZero(INT i) {
        i := 0
    }
] DO {
    OBJ Intbox ib
    ib := Intbox(1)
    CALL setZero(ib)
    PRINTI ib.i
    PRINTLNS ""

    INT i
    i := 1
    CALL setZero(i)
    PRINTI i
    PRINTLNS ""
}
```

Listing 3.6: Example program `intbox1.olang`

The first `setZero` procedure takes as input an `Intbox` and sets its field to 0. The second procedure takes an integer parameter and sets it to 0. In the main program, an `Intbox` and an integer variable, both initialized with value 1, are used as parameters to the procedures. One can observe that as a *side effect* of the `Intbox`-procedure, the parameter object changes. The `INT`-procedure however **does not** change the parameter's value from the main program's point of view. Why do the procedures behave this way? As mentioned, invoking a procedure involves copying the values of the supplied argument expressions. But an object variable holds merely an address, so what happens when object variables are passed as arguments? Technically, object variables are not treated differently from integer variables in this regard, but the value of an object variable is an address, and the value of an integer variable is an integer, so what is copied is an address instead of an integer. This behaviour resembles that of languages like C or Java, where the distinction between *call by value*, *call by reference* and sometimes even *call by sharing* is made. However, this distinction has lead to much confusion because it suggests that the language operates differently for different kinds of variables, while under this, arguably simpler interpretation, it really does not. O employs a strict *call by value* strategy, where the value of an integer variable is an integer, and the value of an object variable is an *address*.

To operate on objects of given classes, instead of procedures, *methods* can be used. A method is a kind of procedure that is defined in the context of a class, and that can be invoked given any object of that class. This given object is not named explicitly as a formal parameter of the method - rather, a method is invoked by using dot notation on object variables, similar to field references. In the method code, the object can be referenced by using `this` - in the same way that is used for initializer code. To illustrate this, the prior example 3.6 could have used a method instead of the `Intbox`-procedure:

```
USING [
    CLASS Intbox(INT i)
    FIELDS INT i
    INIT { this.i := i }
    [
        METHOD setZero() {
            this.i := 0
        }
```

```
    ]

    PROCEDURE setZero(INT i) {
        i := 0
    }
] DO {
    OBJ Intbox ib
    ib := Intbox(1)
    CALL ib.setZero()
    PRINTI ib.i
    PRINTLNS ""

    INT i
    i := 1
    CALL setZero(i)
    PRINTI i
    PRINTLNS ""
}
```

Listing 3.7: Example program `intbox2.olang`

Of course, classes can be used to represent meaningful data structures. Program 3.8 implements rational numbers along with a few operations on them. It asks for two rational numbers which are then added, subtracted, multiplied and divided. The result is provided in simplified form.

```
USING [
    CLASS Rational (INT numerator, INT denominator)
    FIELDS INT numerator
           INT denominator
    INIT {
        IF denominator = 0 THEN {
            PRINTLNS "denominator cannot be zero!"
            ERROR
        }
        this.numerator := numerator
        this.denominator := denominator
    }
    [
        METHOD getNumerator() RETURNS INT num {
            num := this.numerator
        }

        METHOD getDenominator() RETURNS INT den {
            den := this.denominator
        }

        METHOD add(OBJ Rational summand) RETURNS OBJ Rational sum {
            INT newnum
            newnum := this.numerator * summand.getDenominator() + summand.getNumerator()
 ↪  * this.denominator
            INT newden
            newden := this.denominator * summand.getDenominator()
            sum := Rational(newnum, newden)
        }

        METHOD subtract(OBJ Rational subtrahend) RETURNS OBJ Rational difference {
            OBJ Rational addend
            addend := Rational(-subtrahend.getNumerator(), subtrahend.getDenominator())
```

```
            difference := this.add(addend)
        }

        METHOD multiply(OBJ Rational factor) RETURNS OBJ Rational product {
            INT newnum
            newnum := this.numerator * factor.getNumerator()
            INT newden
            newden := this.denominator * factor.getDenominator()
            product := Rational(newnum, newden)
        }

        METHOD divide(OBJ Rational divisor) RETURNS OBJ Rational quotient {
            OBJ Rational reciprocal
            reciprocal := Rational(divisor.getDenominator(), divisor.getNumerator())
            quotient := this.multiply(reciprocal)
        }

        METHOD isPositive() RETURNS INT isPositive {
            isPositive := 1
            IF this.numerator / this.denominator < 0 THEN isPositive := 0
        }

        METHOD isNatural() RETURNS INT isNatural {
            isNatural := 0
            IF this.isPositive() = 1 THEN {
                IF (this.numerator / this.denominator) * this.denominator = this.numerator THEN isNatural := 1
            }
        }

        METHOD simplify() RETURNS OBJ Rational simple
        USING [
            PROCEDURE gcd(INT a, INT b) RETURNS INT res {
                IF a < 0 THEN res := - gcd(-a, b)
                IF NOT a < 0 THEN {
                    IF b < 0 THEN res := - gcd(a, -b)
                    IF b = 0 THEN res := a
                    IF b > 0 THEN {
                        IF a = 0 THEN res := b
                        IF NOT a = 0 THEN {
                            IF a > b THEN res := gcd(a - b, b)
                            IF NOT a > b THEN res := gcd(a, b - a)
                        }
                    }
                }
            }
        ]
        {
            INT gcd
            gcd := gcd(this.numerator, this.denominator)
            simple := Rational(this.numerator / gcd, this.denominator / gcd)
        }

        METHOD compare(OBJ Rational other) RETURNS INT order {
            order := this.numerator * other.getDenominator() - other.getNumerator() * this.denominator
        }

        METHOD print() {
            PRINTI this.numerator
            PRINTS " / "
            PRINTI this.denominator
        }
    ]

```

```
    PROCEDURE readRational() RETURNS OBJ Rational rat {
        PRINTS "Please enter the numerator: "
        INT num
        READ num
        PRINTS "Please enter the denominator: "
        INT den
        READ den
        rat := Rational(num, den)
    }
] DO {
    PRINTLNS "This program prompts you to enter two rational numbers, and performs some
 ↪  calculations with them."
    PRINTLNS "*First number*"
    OBJ Rational fst
    fst := readRational()
    PRINTLNS "*Second number*"
    OBJ Rational snd
    snd := readRational()

    PRINTS "("
    CALL fst.print()
    PRINTS ") + ("
    CALL snd.print()
    PRINTS ") = "
    OBJ Rational sum
    sum := fst.add(snd)
    sum := sum.simplify()
    CALL sum.print()
    PRINTLNS ""

    PRINTS "("
    CALL fst.print()
    PRINTS ") - ("
    CALL snd.print()
    PRINTS ") = "
    OBJ Rational difference
    difference := fst.subtract(snd)
    difference := difference.simplify()
    CALL difference.print()
    PRINTLNS ""

    PRINTS "("
    CALL fst.print()
    PRINTS ") * ("
    CALL snd.print()
    PRINTS ") = "
    OBJ Rational product
    product := fst.multiply(snd)
    product := product.simplify()
    CALL product.print()
    PRINTLNS ""

    PRINTS "("
    CALL fst.print()
    PRINTS ") / ("
    CALL snd.print()
    PRINTS ") = "
    OBJ Rational quotient
    quotient := fst.divide(snd)
    quotient := quotient.simplify()
    CALL quotient.print()
    PRINTLNS ""
}
```

Listing 3.8: Example program `rational.olang`

Running the program with rational numbers $\frac{3}{5}$ and $\frac{7}{9}$ yields:

```
This program prompts you to enter two rational numbers, and performs
 ↪  some calculations with them.
*First number*
Please enter the numerator: 3
Please enter the denominator: 5
*Second number*
Please enter the numerator: 7
Please enter the denominator: 9
(3 / 5) + (7 / 9) = 62 / 45
(3 / 5) - (7 / 9) = 8 / -45
(3 / 5) * (7 / 9) = 7 / 15
(3 / 5) / (7 / 9) = 27 / 35
```

3.5 Inheritance and types

To incorporate all important features of object-oriented programming, O also supports *inheritance*. Inheritance introduces a transitive relation between an *upper* class and a *sub*class. For classes S and U:

Definition 5: ***inheritance relation***

$S \subset U :\Leftrightarrow$ S is a subclass of U

In O, a class can be denoted as a subclass of another class by using the keyword `SUBCLASSOF`, followed by the upper class name, in the class definition. This results in the subclass inheriting the fields and methods of the upper class. The initializer is never inherited, it needs to be redefined in each subclass separately. An inherited method can also be overridden by re-declaring it in the subclass - the re-declared method must have the same name and formal parameter list - except that the formal parameter names are allowed to be different. If the overridden method has no return parameter, the overriding method must not have a return parameter either. If the overridden method does have a return parameter, the overriding method must also have a return parameter, although it is admitted for the return parameter in the overriding method to be of any *subtype* (see next paragraph) of the overridden method's return parameter.

As mentioned, declaring a class introduces a new type to the program. A type that occurs in a program is always one of:

- Type `INT` for integer values, variables and parameters,
- type `BOOL` for boolean values which occur only in conditions,
- and type `OBJ T` for addresses, variables and parameters for objects of class `T`

Without inheritance, the introduced types are all independent of each other. The inheritance relation changes this, giving rise to a *subtype*-relation $<:$, where $S <: T$ reads "S is a subtype of T":

Definition 6: ***subtype relation***

$S <: T :\Leftrightarrow S = T \vee \text{class}(S) \subset \text{class}(T)$, where class(`OBJ C`) = `C`

The subtype relation is a weak partial order on the set of types, because it is reflexive, antisymmetric and transitive.

The principle of *subtype polymorphism* hinted at with return parameters, applies in a much more general way. In O, everywhere a value of type T is expected, a value of type S can be substituted if $S <: T$.

3.5.1 Dynamic binding

One important consequence of subtype polymorphism in O is *dynamic binding* on method calls. Consider the following example 3.9 involving different kinds of animals.

```
USING [
    CLASS Animal()
    INIT {
        PRINTLNS "An animal was born!"
    }
    [
        METHOD makeSound() {
            PRINTLNS "*generic animal sound*"
        }
    ]

    CLASS Dog()
    SUBCLASSOF Animal
    INIT {
        PRINTLNS "A dog was born!"
    }
    [
        METHOD makeSound() {
            PRINTLNS "Woof!"
        }
    ]

    CLASS Cat()
    SUBCLASSOF Animal
    INIT {
        PRINTLNS "A cat was born!"
    }
    [
        METHOD makeSound() {
            PRINTLNS "Meow!"
        }
    ]
] DO {
    INT choice
    OBJ Animal chosenOne

    PRINTLNS "What kind of animal do you like most?"
    PRINTLNS "0: Dogs"
    PRINTLNS "1: Cats"
    PRINTLNS "otherwise: a different one"

    READ choice
    IF choice = 0 THEN {
        PRINTLNS "Congratulations, you get a dog!"
        chosenOne := Dog()
    }
    IF choice = 1 THEN {
        PRINTLNS "Congratulations, you get a cat!"
        chosenOne := Cat()
    }
    IF NOT choice = 0 THEN {
        IF NOT choice = 1 THEN {
            PRINTLNS "Congratulations, you get some other animal!"
            chosenOne := Animal()
        }
    }
    PRINTLNS "What sound does it make?"
    CALL chosenOne.makeSound()
}
```

Listing 3.9: Example program `animals.olang`

Any of the assignments in lines 45, 49 and 54 is valid according to subtype polymorphism, since the resulting object is always of subtype of the variable `chosenOne`. Still, a question arises in line 58. Since both `Cat` and `Dog` override `makeNoise()` from `Animal`, and the behaviour of a variable in the program must reflect its content, the method that is called must be different in each case. This is indeed true. Running the program with input 0 yields:

```
What kind of animal do you like most?
0: Dogs
1: Cats
otherwise: a different one
0
Congratulations, you get a dog!
A dog was born!
What sound does it make?
Woof!
```

Input 1 results in expectedly different behaviour:

```
What kind of animal do you like most?
0: Dogs
1: Cats
otherwise: a different one
1
Congratulations, you get a cat!
A cat was born!
What sound does it make?
Meow!
```

This means that the method that is actually called can not be determined from the type of the variable alone, since it is not possible to know at compile-time about the user input that will be given, or about the results of all computations that may be involved - it can not be *statically bound*. It depends on the type of the object that is referred to by the variable at run-time - this is called *dynamic binding*, and it has far-reaching consequences for the implementation of the code generator and abstract machine discussed in chapter 7 and chapter 6.

To illustrate the utility of inheritance, one might think about implementing a slightly more advanced calculator than example 3.8 which allows for defining arithmetic expressions and evaluating them. An arithmetic expression could be an integer, or a sum, difference, product or quotient. Example 3.10 features a simplified version of the object-oriented composite pattern (see [6]). It uses the classes

- `AExpression` in place of the composite interface (together with `UnaryAExpression` and `BinaryAExpression`),
- `Atom` in place of the leaf class,
- `Sum`, `Difference`, `Product`, `Quotient`, `Faculty` and `Exponential` as composites.

Note that unfortunately, the (invalid) instantiation of one of the interface classes is not checked by the compiler, and hence leads to a run-time error, since O does not support the declaration of interfaces or abstract classes (both of which are a kind of class that must not be instantiated).

```
1 USING [
2     CLASS AExpression()
3     INIT {
```

```
        PRINTLNS "ERROR: This is an interface"
        ERROR
    }
    [
        METHOD evaluate() RETURNS INT res {
            PRINTLNS "ERROR: This is an interface"
            ERROR
        }

        METHOD print() {
            PRINTLNS "ERROR: This is an interface"
            ERROR
        }
    ]

    CLASS UnaryAExpression()
    SUBCLASSOF AExpression
    FIELDS OBJ AExpression inner
    INIT {
        PRINTLNS "ERROR: This is an interface"
        ERROR
    }

    CLASS BinaryAExpression()
    SUBCLASSOF AExpression
    FIELDS OBJ AExpression left
           OBJ AExpression right
    INIT {
        PRINTLNS "ERROR: This is an interface"
        ERROR
    }

    CLASS Atom(INT number)
    SUBCLASSOF AExpression
    FIELDS INT number
    INIT {
        this.number := number
    }
    [
        METHOD evaluate() RETURNS INT num {
            num := this.number
        }

        METHOD print() {
            PRINTI this.number
        }
    ]

    CLASS Sum(OBJ AExpression left, OBJ AExpression right)
    SUBCLASSOF BinaryAExpression
    INIT {
        this.left := left
        this.right := right
    }
    [
        METHOD evaluate() RETURNS INT sum {
            OBJ AExpression left
            OBJ AExpression right
            left := this.left
            right := this.right
            sum := left.evaluate() + right.evaluate()
        }

        METHOD print() {
            OBJ AExpression left
```

```
            OBJ AExpression right
            left := this.left
            right := this.right

            PRINTS "("
            CALL left.print()
            PRINTS " + "
            CALL right.print()
            PRINTS ")"
        }
    ]

    CLASS Difference(OBJ AExpression left, OBJ AExpression right)
    SUBCLASSOF BinaryAExpression
    INIT {
        this.left := left
        this.right := right
    }
    [
        METHOD evaluate() RETURNS INT diff {
            OBJ AExpression left
            OBJ AExpression right
            left := this.left
            right := this.right
            diff := left.evaluate() - right.evaluate()
        }

        METHOD print() {
            OBJ AExpression left
            OBJ AExpression right
            left := this.left
            right := this.right

            PRINTS "("
            CALL left.print()
            PRINTS " - "
            CALL right.print()
            PRINTS ")"
        }
    ]

    CLASS Product(OBJ AExpression left, OBJ AExpression right)
    SUBCLASSOF BinaryAExpression
    INIT {
        this.left := left
        this.right := right
    }
    [
        METHOD evaluate() RETURNS INT product {
            OBJ AExpression left
            OBJ AExpression right
            left := this.left
            right := this.right
            product := left.evaluate() * right.evaluate()
        }

        METHOD print() {
            OBJ AExpression left
            OBJ AExpression right
            left := this.left
            right := this.right

            PRINTS "("
            CALL left.print()
            PRINTS " * "
```

```
            CALL right.print()
            PRINTS ")"
        }
    ]

    CLASS Quotient(OBJ AExpression dividend, OBJ AExpression divisor)
    SUBCLASSOF BinaryAExpression
    INIT {
        this.left := dividend
        this.right := divisor
    }
    [
        METHOD evaluate() RETURNS INT product {
            OBJ AExpression left
            OBJ AExpression right
            left := this.left
            right := this.right
            INT divisor
            divisor := right.evaluate()
            IF divisor = 0 THEN {
                PRINTLNS "ERROR: divisor must not be zero."
                ERROR
            }
            product := left.evaluate() / divisor
        }

        METHOD print() {
            OBJ AExpression left
            OBJ AExpression right
            left := this.left
            right := this.right

            PRINTS "("
            CALL left.print()
            PRINTS " / "
            CALL right.print()
            PRINTS ")"
        }
    ]

    CLASS Faculty(OBJ AExpression inner)
    SUBCLASSOF UnaryAExpression
    INIT {
        this.inner := inner
    }
    [
        METHOD evaluate()
        RETURNS INT faculty
        USING [
            PROCEDURE faculty(INT num) RETURNS INT faculty {
                IF num < 0 THEN {
                    PRINTLNS "ERROR: Undefined factorial"
                    ERROR
                }
                IF num = 0 THEN {
                    faculty := 1
                }
                IF num > 0 THEN {
                    faculty := num * faculty(num - 1)
                }
            }
        ] {
            OBJ AExpression inner
            inner := this.inner
            INT num
```

```
            num := inner.evaluate()
            faculty := faculty(num)
        }

        METHOD print() {
            OBJ AExpression inner
            inner := this.inner

            PRINTS "("
            CALL inner.print()
            PRINTS "!)"
        }
    ]

    CLASS Exponential(OBJ AExpression base, OBJ AExpression exponent)
    SUBCLASSOF BinaryAExpression
    INIT {
        this.left := base
        this.right := exponent
    }
    [
        METHOD evaluate()
        RETURNS INT exp
        USING [
            PROCEDURE exp(INT base, INT exponent) RETURNS INT exp {
                IF exponent < 0 THEN {
                    PRINTLNS "ERROR: Illegal exponent"
                    ERROR
                }
                IF base = 0 THEN {
                    exp := 0
                    IF exponent = 0 THEN {
                        exp := 1
                    }
                }
                IF NOT base = 0 THEN {
                    IF exponent = 0 THEN {
                        exp := 1
                    }
                    IF exponent > 0 THEN {
                        exp := base * exp(base, exponent - 1)
                    }
                }
            }
        ] {
            OBJ AExpression base
            base := this.left
            OBJ AExpression exponent
            exponent := this.right
            INT b
            b := base.evaluate()
            INT e
            e := exponent.evaluate()
            exp := exp(b, e)
        }

        METHOD print() {
            OBJ AExpression left
            OBJ AExpression right
            left := this.left
            right := this.right

            PRINTS "("
            CALL left.print()
            PRINTS "^"
```

```
264             CALL right.print()
265             PRINTS ")"
266         }
267     ]
268
269 ] DO {
270     OBJ AExpression exp
271     exp := Product(Exponential(Faculty(Atom(3)), Exponential(Atom(3), Atom(3))),
    ↪   Quotient(Product(Atom(3), Atom(4)), Difference(Atom(9), Atom(7))))
272
273     CALL exp.print()
274     PRINTS " = "
275     PRINTI exp.evaluate()
276 }
```

Listing 3.10: Example program `expression.olang`

Note the deliberate use of subtype polymorphism and dynamic binding in lines 271, 273 and 275, as well as the recursive method calls. Instead of entering the expression through the standard input, it is directly defined in the program. Running the program yields the following output:

```
(((3!)^(3^3)) * ((3 * 4) / (9 - 7))) = 6140942214464815497216
```

3.5.2 Static binding and invocation ambiguity

While method invocations are dynamically bound, the formal parameters of methods and procedures are statically bound in O. To understand what static binding means, and how it differs from dynamic binding, it might be useful to consider example 3.9 once again. In examples 3.6 and 3.7, it was demonstrated how a procedure could be substituted by a method. Once inheritance is involved, this substitution becomes problematic:

```
USING [
    CLASS Animal()
    INIT {
        PRINTLNS "An animal was born!"
    }
    [
        METHOD makeSound() {
            PRINTLNS "*generic animal sound*"
        }
    ]

    CLASS Dog()
    SUBCLASSOF Animal
    INIT {
        PRINTLNS "A dog was born!"
    }
    [
        METHOD makeSound() {
            PRINTLNS "Woof!"
        }
    ]

    CLASS Cat()
    SUBCLASSOF Animal
    INIT {
        PRINTLNS "A cat was born!"
    }
    [
        METHOD makeSound() {
            PRINTLNS "Meow!"
        }
```

```
    ]

    PROCEDURE makeSound(OBJ Animal a) {
        PRINTLNS "*generic animal sound*"
    }

    PROCEDURE makeSound(OBJ Dog d) {
        PRINTLNS "Woof!"
    }

    PROCEDURE makeSound(OBJ Cat c) {
        PRINTLNS "Meow!"
    }
] DO {
    OBJ Animal a
    a := Animal()
    CALL makeSound(a)
    CALL a.makeSound()
    a := Dog()
    CALL makeSound(a)
    CALL a.makeSound()
    a := Cat()
    CALL makeSound(a)
    CALL a.makeSound()
}
```

Listing 3.11: Example program `animals-procedures.olang`

One might think that the invocations `makeSound(a)` and `a.makeSound()` would result in the same output given that the defined procedures mirror the methods above. But in fact, the effect is completely different, as the output shows:

```
An animal was born!
*generic animal sound*
*generic animal sound*
A dog was born!
*generic animal sound*
Woof!
A cat was born!
*generic animal sound*
Meow!
```

The second and third invocations produce different outputs since formal parameters are *statically bound*. This static binding uses only syntactic information about the involved types that is available at compile-time. In this case, the compiler knows that the variable `a` is of type `OBJ Animal` - but not about the exact type of the involved values at run-time, as shown in 3.9.

Implementing dynamic binding for formal parameters is possible, but not usually done for two reasons. Firstly, the implementation is relatively slow and complicated - invocations rely on runtime type information and extensive checks must be performed before every invocation. Secondly, it can make program execution more unpredictable, especially once parameter lists get longer and more overloaded procedures or methods (that is, multiple declarations with same name but different formal parameter lists) are involved. This holds especially true for mini-languages where simplicity and clarity is of importance.

To find the correct procedure or method to invoke, the compiler first calculates the set of methods or procedures that are applicable for the given invocation, and then picks the most specific match (if a most specific match exists). To see why this process is not as simple as it seems, and that unexpected failure modes exist, a more precise notion of *applicable* and *most specific* is needed.

The first step is to generalize the subtype relation to sequences of types.

Definition 7: ***applicability relation***

A finite sequence S of types is *applicable* to another sequence T of types if both share the same length and any two respective members are related through the subtype relation:

$$(S)_{i=0}^{n} <: (T)_{i=0}^{m} :\Leftrightarrow n = m \wedge \forall 0 \leq i \leq n.\ S_i <: T_i$$

The reflexivity, antisymmetry and transitivity of the applicability relation follows from the subtype relation's properties, hence the applicability relation is also a weak partial order.

An invocation then matches a method or procedure if the name matches and the invocation's sequence of compile-time parameter expression types is applicable to the sequence of types of the formal parameter list. Formally, the set of matching procedures or methods can be defined:

Definition 8: ***matching set***

Given the set C of procedures or methods in question and an invocation of name N, with sequence T of expression types, the set of matching type sequences or *matching set* M is

$$M = \{\ T' \mid \exists p \in C.\ N = name(p) \wedge T' = types(p) \wedge T <: T'\ \}$$

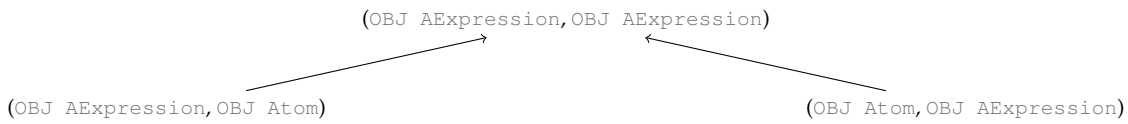
The *name* and *types* functions denote the corresponding procedure or method name and type sequence.

The matching set forms a weak partially ordered set together with the applicability relation. The most specific match is the procedure or method whose formal parameter list corresponds to the minimum of the matching set. In case there is no minimum, the matching set is either empty or ambiguous.

To illustrate this, consider again example 3.10 of arithmetic expressions, but with some additional procedure definitions in the preamble:

```
    ...
    PROCEDURE foo(OBJ AExpression e, OBJ AExpression e) { ... }
    PROCEDURE foo(OBJ AExpression e, OBJ Atom a) { ... }
    PROCEDURE foo(OBJ Atom a,        OBJ AExpression e) { ... }
] DO {
    OBJ Atom one
    OBJ Atom two
    ...
    CALL foo(one, two)
}
```

The invocation's matching set is



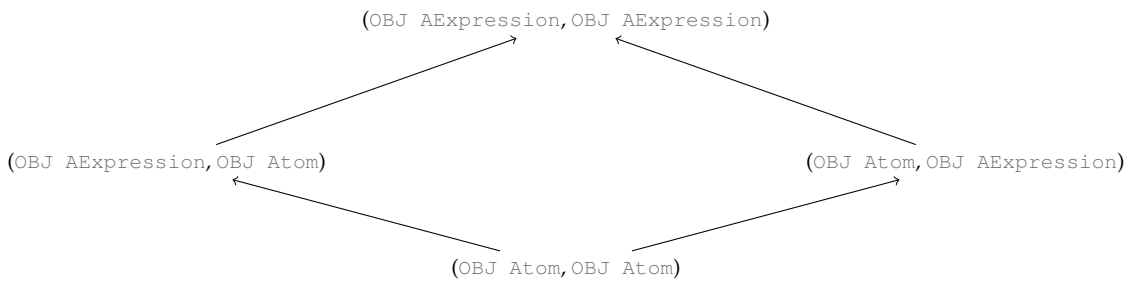
Since the set has no minimum, it is ambiguous and the invocation fails at compile-time.

The problem can be alleviated by removing one of the procedures or adding a fourth:

```
    ...
    PROCEDURE foo(OBJ AExpression e, OBJ AExpression e) { ... }
```

```
    PROCEDURE foo(OBJ AExpression e, OBJ Atom a) { ... }
    PROCEDURE foo(OBJ Atom a,        OBJ AExpression e) { ... }
    PROCEDURE foo(OBJ Atom aOne,     OBJ Atom aTwo) { ... }
] DO {
    OBJ Atom one
    OBJ Atom two
    ...
    CALL foo(one, two)
}
```

With this fourth procedure, the matching set is



which results in a successful invocation of procedure `foo(OBJ Atom aOne, OBJ Atom aTwo)`.

3.5.3 Liskov substitution principle

The notion of subtype polymorphism introduced for O is purely syntactic in nature. By inheritance, it is guaranteed that a subclass always exhibits the superclass's interface. This is sufficient for ensuring that the program is type-correct in the sense that no sudden crash can occur due to missing methods or incompatible return values, but it cannot ensure that the program works correctly for any given subtype. This is because general correctness is a semantic property - it depends on the behaviour of the objects. The *Liskov substitution principle* or *subtype requirement* (see [9]) formalizes this:

Definition 9: ***subtype requirement***

Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S if S $<:$ T.

This notion of behavioural subtyping is more meaningful, but it goes beyond the scope of simple type checking. The programmer has to ensure correct behaviour using formal methods, some of which are covered in [9].

As a simple example of a violation of the subtype requirement, consider once again the class for rational numbers from example 3.8 and consider this override of method `add` in a subclass `Integer`:

```
CLASS Integer
SUBCLASSOF Rational
...
[
    ...
```

```
    METHOD add(OBJ Rational summand) RETURNS OBJ Rational sum {
        sum := this
    }
    ...
]
```

Even though such a program would be type-correct, it obviously violates the subtype requirement since the behaviour should be calculating the sum of both numbers, not the identity of the first number.

3.6 Formal syntax

To conclude the introduction of the language O and prepare for the implementation in the latter part of the thesis, a formal syntax definition for O is needed. The first step of defining the lexical syntax was already done in section 3.1. This provides the lexemes that the actual formal grammar is built upon. The following is a formal grammar described in W3C-EBNF-syntax:

```
Program                 ::= ('USING' '[' ClassDeclaration* ProcedureDeclaration* ']')?
                            'DO' Command
ClassDeclaration        ::= 'CLASS' ClassName FormalParameterList
                            ('SUBCLASSOF' ClassName)?
                            ('FIELDS' SymbolDeclaration+)?
                            'INIT' Command
                            ('[' MethodDeclaration+ ']')?
IntSymbolDeclaration    ::= 'INT' SymbolName
ObjectSymbolDeclaration ::= 'OBJ' ClassName SymbolName
SymbolDeclaration       ::= IntSymbolDeclaration | ObjectSymbolDeclaration
FormalParameterList     ::= '(' (SymbolDeclaration (',' SymbolDeclaration)*)? ')'
ActualParameterList     ::= '(' (Expression (',' Expression)*)? ')'
MethodDeclaration       ::= 'METHOD' ProcedureHeader Command
ProcedureDeclaration    ::= 'PROCEDURE' ProcedureHeader Command
ProcedureHeader         ::= SymbolName FormalParameterList
                            ('RETURNS' SymbolDeclaration)?
                            ('USING' '[' ProcedureDeclaration+ ']')?
Call                    ::= SymbolReference ActualParameterList?
SymbolReference         ::= SymbolName ('.' SymbolName)?
Command                 ::= SymbolReference ':=' Expression
                          | SymbolDeclaration
                          | 'CALL' Call
                          | 'READ' SymbolName
                          | '{' Command+ '}'
                          | 'IF' Condition 'THEN' Command
                          | 'WHILE' Condition 'DO' Command
                          | 'PRINTI' Expression
                          | 'PRINTS' String
                          | 'PRINTLNS' String
                          | 'ERROR'
Condition               ::= Expression Relation Expression | 'NOT' Condition
Relation                ::= '=' | '<' | '>'
Expression              ::= ('+' | '-')? Term (('+' | '-') Term)*
Term                    ::= Factor (( '*' | '/' ) Factor)*
Factor                  ::= Call
                          | ClassName ActualParameterList  /* Class Instantiation */
                          | Integer
                          | '(' Expression ')'
```

Listing 3.12: Formal EBNF grammar for the language O

Since chapter 5 implements an LL(1)-Parser, an LL(1)-grammar is needed. Such a grammar can be obtained from the above by removing the +, * and ? operators and simulating them with new

non-terminals. This must be done by relying on right-recursion instead of left-recursion. To check that the resulting grammar is of type LL(1), the two LL(1)-conditions (see [3]) must be verified by calculating the corresponding $first$- and $follow$-sets and verifying that they are disjoint. A transformed grammar, suitable for applying it to the online context free grammar checker (see [11]), can be found in the provided implementation's file `resources/syntax/syntax.grammar-checker`.

Note that the LL(1)-property relies on the lexeme abstraction. Otherwise, lines 29, 30 and 31 alone would destroy the LL(1)-property due to them all beginning with the character '`P`', creating overlapping $first$-sets for the `Command`-rule. But since they are treated as the atomic lexemes `PRINTI`, `PRINTS` and `PRINTLNS`, the sets are disjoint.

CHAPTER 4

Concepts of Compilers

The concepts used in the implementation of O are largely based on adaptations of the ideas for implementing imperative languages presented in [3]. The language is translated by a compiler with two main components. The first component is the *parser* described in chapter 5, which utilizes the underlying lexical syntax and formal grammar defined in section 3.1 and section 3.6 to create an abstract tree representation of the program. This representation is then used by the *code generator* described in chapter 7 to generate the machine code. Instead of a physical machine, the translation target is an *abstract machine* that is simulated by a Haskell program. This program is implemented as a separate component described in chapter 6.

The approach of using abstract machines provides two major benefits. Firstly, using a simulated machine makes a program portable to anywhere this machine can run. This obviates the need for far-reaching abstractions like intermediate-code generation (see [1]) to support multiple machine architectures with the same compiler. Secondly, since an abstract machine is not bound to any physical machine model, it can be designed to match the language's memory model and type system very well. Both combined result in a considerable simplification of the code generator, at the relatively lower cost of introducing a new, but independent component to the system. This has helped projects like the Java Virtual Machine to great success - today, in addition to Java, there are numerous languages compiled to JVM bytecode. And it is also the main reason for pursuing the abstract machine approach for the implementation of O.

CHAPTER 5

Tokenizer and Parser Implementation for O

The parsing process of the language consists of two separate phases: the *tokenization* or *lexing* phase and the actual *parsing* phase. For each phase, there is a separate component that realizes the corresponding functionality. In the following sections, both of them will be discussed in greater detail.

5.1 Tokenizer

The tokenizer realizes a mapping from an input sequence of characters, which represents the program text, to a list of abstract tokens. The abstract tokens represent the keywords, symbols and identifiers that can occur in a program. For example, the tokenizer would map the input `":="` to a token that could be called `(::=)`. Some of these abstract tokens can carry additional information as well, take the input `"Rational"` as an example. This is not a keyword or symbol of our language, but it is a valid class identifier. The corresponding abstract token has to carry this name with it, so the tokenizer could map this input to `ClassName "Rational"`. The input `"i := i + 1"` would then be mapped to the list `[SymbolName "i", (::=), SymbolName "i", (:+), Integer 1]`.

In other words, the tokenizer receives as input a sequence of characters and returns a sequence of tokens. It can be implemented as a simple backtracking parser that produces a corresponding token if the input matches a given mapping from keywords to abstract tokens. The parsing library `parsec` [1] happens to lend itself well to this kind of task, so it is also being used for the tokenizer. This token-parser needs the capability to backtrack in case some tokens' string representations share a non-empty prefix, which is the case for this language - amongst others, there are both the tokens `PROGRAM` and `PROCEDURE`. To incorporate backtracking, parsec's *try*-operator is used.

5.2 Parser

The parser fulfils 3 functions in the compiling process:

1. Ensuring that the program is syntactically correct,

[1] Parsec is a parser combinator library (formerly called combinator parsers) that allows the construction of complex parsers from simple ones. For more information about the library, see [8]. To ease the understanding of the token-parser's code, it might be helpful to look at the parser (see 5.2) first.

2. Creating a data structure that holds all information the code generator (see chapter 7) requires in order to generate the program's machine code (see chapter 6),

3. Discarding information like braces or separators, that the code generator does not need.

To accomplish this, it first derives the input from the target grammar. As was already established in chapter 3, the grammar is of type LL(1). Both LL(1)-Conditions together imply that the derivation process can be defined in a deterministic way, since at any particular point during a leftmost derivation, there is at most one applicable production that matches the input (compare [3]). Therefore, backtracking is not needed.

The parser is implemented as a *non-backtracking recursive descent parser*. A *recursive descent parser* performs the derivation recursively, defining parsers not only for the start symbol, but for any non-terminal. These parsers then recursively run each other and consume the prefix of the input stream that belongs to their respective non-terminal.

This recursive structure makes it simple to construct the *Abstract Syntax Tree (AST)* directly as part of the derivation process. The AST holds all information that is necessary for generating the machine code.

Example: Parsing expressions and comparisons

Suppose one needs to parse the comparison `"1 + 1 = 2"`. The resulting abstract token list is `[Integer 1, (:+), Integer 1, (:=), Integer 2]`. In the provided implementation, these are the `data`-definitions for comparisons and expressions, with the relevant lines highlighted:

```
data Condition
  = Comparison Expression Relation Expression
  | Negation Condition

data Relation = Equals | Smaller | Greater

data Expression
  = Expression (NonEmpty (Sign, Term))

data Term
  = Term Factor [(Operator, Factor)]

data Factor
  = CallFactor Call
  | ClassInstantiation ClassName ActualParameterList
  | Integer Integer
  | CompositeFactor Expression

data Sign = Plus | Minus

data Operator = Times | Divide
```

The corresponding parser code that generates these structures is:

```
condition =
  (Comparison <$> expression <*> relation <*> expression)
    <|> (Negation <$> (accept NOT *> condition))

relation =
```

```
  (accept (:<) *> pure Smaller)
    <|> (accept (:>) *> pure Greater)
    <|> (accept (:=) *> pure Equals)

expression =
  Expression <$> ((:|) <$> firstSignTerm <*> manySignTerms)
  where
    firstSignTerm = do
      ms <- optionMaybe sign
      t <- term
      return (fromMaybe Plus ms, t)
    manySignTerms = many ((,) <$> sign <*> term)

term = Term <$> factor <*> many ((,) <$> operator <*> factor)

factor =
  (CallFactor <$> call)
    <|> (ClassInstantiation
          <$> acceptClassName
          <*> actualparameterlist)
    <|> (Integer <$> acceptInteger)
    <|> (CompositeFactor <$>
          (accept OpenRoundBracket
            *> expression
            <* accept CloseRoundBracket))

sign =
  (accept (:-) *> pure Minus)
    <|> (accept (:+) *> pure Plus)

operator =
  (accept (:*) *> pure Times)
    <|> (accept (:/) *> pure Divide)
```

Above code warrants further explanation. The provided implementation uses the parser combinator library `parsec` (see [8]) which relies heavily on various instances of applicative functors and monads. Notably, parsers defined in parsec are automatically an applicative and monadic action. The code `Comparison <$> expression <*> relation <*> expression` will create a parser that takes the resulting expressions and relation in order, and wraps them into a `Comparison`. The same can be accomplished with monadic notation as well:

```
do
  e1 <- expression
  r <- relation
  e2 <- expression
  return $ Comparison e1 r e2
```

The provided implementation uses a combination of applicative notation in most cases and monadic notation in cases where the structure of the syntax tree diverges more from the formal grammar.

The Alternative-operator `(<|>)` implements branching:

```
sign =
  (accept (:-) *> pure Minus)
```

```
<|> (accept (:+) *> pure Plus)
```

In case the next token is a `(:-)`, the parser yields the result of `accept (:-) *> pure Minus`. Otherwise, the next branch is tried. `<|>` is right-associative, which effectively results in parsec evaluating the alternatives in sequential order.

It is worth mentioning that parsec's *try*-operator is absent in the parser implementation, due to the fact that it is *non-backtracking*.

With this in mind, one can see that the token list `[Integer 1, (:+), Integer 1, (:=), Integer 2]` will get parsed into the following syntax tree:

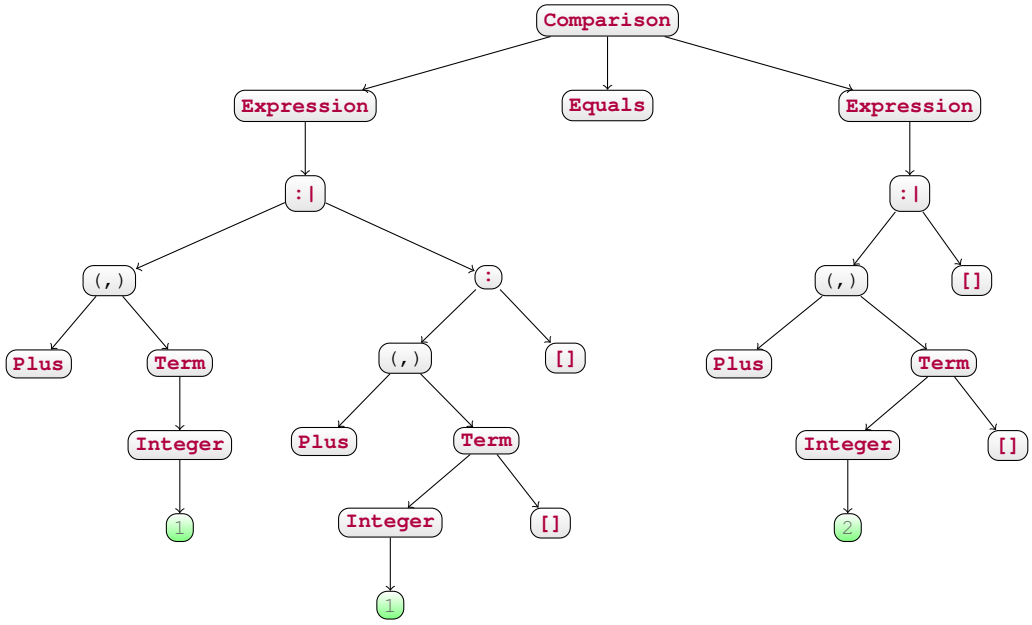


Figure 5.1: syntax tree for comparison "1 + 1 = 2"

5.3 Note on online algorithms and lazy evaluation

In most compilers, the tokenizer is implemented as an *online-algorithm* (see [2]). Translated to the functional paradigm, this would mean that the parser could start evaluating before the tokenizer is finished evaluating its input. One might think that Haskell's *lazy evaluation strategy* (see [5]) could effectively result in the tokenizer being evaluated in this fashion. Unfortunately, independent of the evaluation strategy, the provided implementation does not allow for this property - for the parser to start evaluating, the tokenizer must have already finished evaluating the input. This is due to the fact that `parsec` (which is, as mentioned, also used for the tokenizer) always distinguishes between a successful parse (`Right _`) and a parse error (`Left _`), and even the last input character could still trigger an error. The same argument holds true for the parser in the code generator's context (see chapter 7), which is less relevant, but still interesting to note.

CHAPTER 6

Abstract Machine Implementation for O

The abstract machine for O consists of a set of simple core instructions along with the necessary data structures, which will be introduced in the next section. This core machine supports O's imperative constructs. In the latter sections, the core is extended to support O's procedures and object-oriented features. The extension of the instruction set then serves as the target language of the code generator (see chapter 7). The machine is implemented as a Haskell program.

One feature that is notably absent from both the abstract machine and code generator is *garbage collection*. Garbage collection is a mechanism of reclaiming memory that is occupied by objects that are safe to remove because they are neither directly nor indirectly referenced in the program any more (see [1]). The omission of a garbage collection mechanism would impose an unacceptable restriction for any modern production-grade language supporting object-oriented concepts without explicit memory management. But it is a compromise worth making in order to simplify both components of the provided implementation.

6.1 The core machine

The core machine is *stack-based* - it builds on a set of machine instructions that operate by manipulating the machine's *stack memory* and *registers*. Additionally, certain machine instructions can make the machine ask for an input from the runtime environment or generate some output to the runtime environment of the machine.

The provided implementation defines three runtime environments, one of which is the default environment that connects to the operating system's standard input and output, the other two are for testing and generating traces of the program execution.

For storing the machine instructions to execute, the machine also has a *code memory* `code`, which is simply a sequence of instructions. During the runtime of the machine, this code memory is not changed.

The core machine has two registers. The *instruction register* `I`, which stores the instruction that is currently being executed, and the *program counter* `PC`, which stores the address of the next instruction in the code memory.

The *stack memory* `stack` is a mutable sequence of integer values that is always initialized as `[0, 0]`. This non-empty initial value is to ensure compatibility of machine instructions after the extensions in section 6.2 are applied - otherwise, it can be ignored. In contrast to the code memory, the stack memory can grow or shrink during the runtime of the machine. As the name

suggests, the stack memory is generally being used as a stack, which means that values are either *pushed* onto the top of the stack or they are *popped* (removed) from the top of the stack - the top meaning the rightmost value, or value with the highest address in the sequence. Despite the name suggesting otherwise, some instructions can and will read or write values below the top of the stack.

The core instructions are explained in table 6.1 and 6.2, presented both using an intuitive explanation and pseudocode. The pseudocode uses the following abbreviations:

Abbreviation	**Meaning**
`loadInstruction a`	`I := code[a]` `PC := a + 1`
`pop`	Pop the stack's topmost value, yielding the popped value.
`push i`	Push the value of `i` onto the stack.
`print e`	Print value of expression `e` to the environment output.
`read`	Read an integer value from the environment input, yielding the integer value.
`nop`	Do nothing.

Instruction	**Intuition**	**Pseudocode**
`PushInt` `n`	Push integer `n` onto the stack.	`push n` `loadInstruction PC`
`LoadStack` `a`	Load the value from stack address `a` and push it onto the stack.	`push stack[2 + a]` `loadInstruction PC`
`StoreStack` `a`	Pop the stack's topmost value and store it to stack address `a`.	`stack[2 + a] := pop` `loadInstruction PC`
`CombineUnary` `op`	Combine the stack's topmost value with operator `op`. Supported operators: `NOT` (¬)	`push (op pop)` `loadInstruction PC`

`CombineBinary` `op`	Combine the stack's two topmost values using operator `op`. Supported operators: Equals (=), Smaller (<), Greater (>), Plus (+), Minus (-), Times (*), Divide (/)	`snd := pop`
		`fst := pop`
		`push (op fst snd)`
		`loadInstruction PC`

Table 6.1: The core machine instructions, part 1

Jump a	Unconditionally jump to code address a.	`loadInstruction a`

JumpIfFalse a	Jump to code address a if the stack's topmost value represents a boolean value of **False**.	`if pop = 0`
		`then loadInstruction a`
		`else loadInstruction PC`

Read	Read an integer value from the environment's input and push it onto the stack.	`push read` `loadInstruction PC`
PrintInt	Pop the stack's topmost value and print it to the environment's output.	`print pop` `loadInstruction PC`
PrintStr s	Print the string s to the environment's output.	`print s` `loadInstruction PC`
PrintStrLn s	Print the string s followed by a new line character to the environment's output.	`print (s ++ '\n')` `loadInstruction PC`
Halt	Halt the machine.	`nop`

Table 6.2: The core machine instructions, part 2

Stepping the machine is done by executing one instruction. When a machine program is to be run, a corresponding machine is created, with the registers and memory initialized. Running a machine means stepping it repeatedly until the **Halt**-instruction is reached. Therefore, the instruction cycle can be described by pseudocode 6.1.

```
code := program
I := code[0]
PC := 1
stack := [0, 0]
while I != Halt do executeInstruction
```

Listing 6.1: Instruction cycle for core machine

As an example of a machine program execution, consider program 6.2 which calculates the factorial of a natural number.

```
DO {
    PRINTS "Please enter a natural number n: "
    INT n
    READ n
    INT faculty
```

```
    faculty := 1
    IF n < 0 THEN {
        PRINTI n
        PRINTLNS " is not a natural number!"
        ERROR
    }
    WHILE n > 0 DO {
        faculty := faculty * n
        n := n - 1
    }
    PRINTS "n! = "
    PRINTI faculty
}
```

Listing 6.2: Example program `fac0.olang`

The compiler translates this O-program into the machine program depicted in listing 6.3. Program instructions are annotated with a mix of corresponding elements from O and machine pseudocode to make the program more readable.

```
PushInt 0                   # BEGIN of main program - stack memory allocation for n
PushInt 0                   # stack memory allocation for faculty
PrintStr "Please enter a natural number n: "
PushInt 0
StoreStack 0                # initialize n := 0
Read
StoreStack 0                # READ n
PushInt 0
StoreStack 1                # initialize faculty := 0
PushInt 1
StoreStack 1                # faculty := 1
LoadStack 0                 # push n
PushInt 0
CombineBinary Smaller       # IF n < 0
JumpIfFalse 19              # skip IF body if n >= 0
LoadStack 0                 # push n
PrintInt                    # PRINTI n
PrintStrLn " is not a natural number!"
Halt                        # ERROR
LoadStack 0                 # push n
PushInt 0
CombineBinary Greater       # WHILE n > 0
JumpIfFalse 32              # skip WHILE body if n <= 0
LoadStack 1                 # push faculty
LoadStack 0                 # push n
CombineBinary Times         # push faculty * n
StoreStack 1                # faculty := faculty * n
LoadStack 0                 # push n
PushInt 1
CombineBinary Minus         # push n - 1
StoreStack 0                # n := n - 1
Jump 19                     # end of WHILE body: return to condition
PrintStr "n! = "
LoadStack 1                 # push faculty
PrintInt                    # PRINTI faculty
Halt                        # END of main program
```

Listing 6.3: Annotated machine code for example 6.2

Run with $n = 3$, machine program 6.3 produces the sequence of machine states depicted in table 6.3.

Step	PC	I	Stack
0	1	**PushInt** 0	[0,0]
1	2	**PushInt** 0	[0,0,0]
2	3	**PrintStr** "Please enter a natural number n: "	[0,0,0,0]
3	4	**PushInt** 0	[0,0,0,0]
4	5	**StoreStack** 0	[0,0,0,0,0]
5	6	**Read**	[0,0,0,0]
6	7	**StoreStack** 0	[0,0,0,0,3]
7	8	**PushInt** 0	[0,0,3,0]
8	9	**StoreStack** 1	[0,0,3,0,0]
9	10	**PushInt** 1	[0,0,3,0]
10	11	**StoreStack** 1	[0,0,3,0,1]
11	12	**LoadStack** 0	[0,0,3,1]
12	13	**PushInt** 0	[0,0,3,1,3]
13	14	**CombineBinary Smaller**	[0,0,3,1,3,0]
14	15	**JumpIfFalse** 19	[0,0,3,1,0]
15	20	**LoadStack** 0	[0,0,3,1]
16	21	**PushInt** 0	[0,0,3,1,3]
17	22	**CombineBinary Greater**	[0,0,3,1,3,0]
18	23	**JumpIfFalse** 32	[0,0,3,1,1]
19	24	**LoadStack** 1	[0,0,3,1]
20	25	**LoadStack** 0	[0,0,3,1,1]
21	26	**CombineBinary Times**	[0,0,3,1,1,3]
22	27	**StoreStack** 1	[0,0,3,1,3]
23	28	**LoadStack** 0	[0,0,3,3]
24	29	**PushInt** 1	[0,0,3,3,3]
25	30	**CombineBinary Minus**	[0,0,3,3,3,1]
26	31	**StoreStack** 0	[0,0,3,3,2]
27	32	**Jump** 19	[0,0,2,3]
28	20	**LoadStack** 0	[0,0,2,3]
29	21	**PushInt** 0	[0,0,2,3,2]
30	22	**CombineBinary Greater**	[0,0,2,3,2,0]
31	23	**JumpIfFalse** 32	[0,0,2,3,1]
32	24	**LoadStack** 1	[0,0,2,3]
33	25	**LoadStack** 0	[0,0,2,3,3]
34	26	**CombineBinary Times**	[0,0,2,3,3,2]
35	27	**StoreStack** 1	[0,0,2,3,6]
36	28	**LoadStack** 0	[0,0,2,6]
37	29	**PushInt** 1	[0,0,2,6,2]
38	30	**CombineBinary Minus**	[0,0,2,6,2,1]
39	31	**StoreStack** 0	[0,0,2,6,1]
40	32	**Jump** 19	[0,0,1,6]
41	20	**LoadStack** 0	[0,0,1,6]
42	21	**PushInt** 0	[0,0,1,6,1]
43	22	**CombineBinary Greater**	[0,0,1,6,1,0]
44	23	**JumpIfFalse** 32	[0,0,1,6,1]
45	24	**LoadStack** 1	[0,0,1,6]
46	25	**LoadStack** 0	[0,0,1,6,6]
47	26	**CombineBinary Times**	[0,0,1,6,6,1]
48	27	**StoreStack** 1	[0,0,1,6,6]
49	28	**LoadStack** 0	[0,0,1,6]
50	29	**PushInt** 1	[0,0,1,6,1]
51	30	**CombineBinary Minus**	[0,0,1,6,1,1]
52	31	**StoreStack** 0	[0,0,1,6,0]
53	32	**Jump** 19	[0,0,0,6]
54	20	**LoadStack** 0	[0,0,0,6]
55	21	**PushInt** 0	[0,0,0,6,0]
56	22	**CombineBinary Greater**	[0,0,0,6,0,0]
57	23	**JumpIfFalse** 32	[0,0,0,6,0]
58	33	**PrintStr** "n! = "	[0,0,0,6]
59	34	**LoadStack** 1	[0,0,0,6]
60	35	**PrintInt**	[0,0,0,6,6]
61	36	**Halt**	[0,0,0,6]

Table 6.3: Machine trace for program 6.3, $n = 3$

Note that the manual reproduction of above trace would show the irregularity of **Read** actually taking two steps to execute. The presented trace was obtained using a non-interactive machine environment. All interactive machine environments are initiated with an empty machine input buffer, which causes the machine to request a new input when encountering a **Read**-instruction. This delays the execution by one step. Non-interactive environments like the testing-

environment pre-supply all necessary input in the buffer, so the execution is not delayed.

6.2 Support for procedures

To simplify the code generator, the abstract machine should explicitly support procedure invocations. To understand how this support is to be implemented, it is necessary to consider again how procedure invocations work in O. When a procedure is invoked, parameter values are copied into the context of the procedure. Then, the procedure code execution starts. After that, the program returns to the next instruction after the invocation - possibly returning some value, and discarding the local variables and parameter values of the invocation.

This hints at the possibility of managing the procedure invocations' data on a stack, making it compatible with the structure of the abstract machine. On the machine's stack, the data that represents the context of a procedure invocation is called a *procedure activation record* or *stack frame*. A stack frame contains all information that is needed to execute the procedure's machine instructions correctly for the given invocation. As procedure invocations continue to happen, new stack frames are pushed onto the machine's stack. They are popped when the corresponding invocation ends. Each stack frame has a *base address* that is the index of the first element in the frame. The data in a stack frame consists of:

Data	Description
dynamic link `DL`	The base address of the directly preceding stack frame - which is the one that belongs to the *caller* of the current invocation.
return address `RA`	The address of the instruction in `code` where the execution should continue after the procedure finishes execution.
local variables	A sequence of integer values representing the state of the invocation's parameters, local declarations and return parameter.
local stack	A sequence of integer values that can grow and shrink during procedure execution and that serves as temporary memory for executing O-instructions.

The main program can be thought of having the first stack frame, with an undefined dynamic link and return address. This is not a problem since the program ends with the main program invocation. Table 6.4 displays the stack frame layout followed in the abstract machine, from bottom (index 0) to top.

Relative index	Content
0	`DL`
1	`RA`
$2 .. n+1$	The n local variables
$n+2 ..$	The local stack

Table 6.4: Stack frame layout for abstract machine

To store the base address of the current invocation's stack frame, the machine is extended with a *base address register* `B`.

Since in O, procedures are self-contained in that they cannot refer to external variables, the abstract machine does not need to implement the kind of relative addressing used for accessing *static predecessors* (see [3]). The only kind of relative addressing needed is the addressing of local variables relative to the current base address. Table 6.5 shows the changes necessary to make

the core machine instructions capable of relative addressing. Tables 6.6 and 6.7 show the new machine instructions that are introduced. Additionally, the instruction cycle is slightly altered to reflect the addition of the base address register `B`:

```
code := program
I := code[0]
PC := 1
stack := [0, 0]
B := 0
while I != Halt do executeInstruction
```

Listing 6.4: Instruction cycle for core machine with procedures

Instruction	Intuition	Pseudocode
`LoadStack` `a`	Load the value from relative stack address `a` and push it onto the stack.	`push stack[B + 2 + a]` `loadInstruction PC`
`StoreStack` `a`	Pop the stack's topmost value and store it to relative stack address `a`.	`stack[B + 2 + a] := pop` `loadInstruction PC`

Table 6.5: Necessary adaptations for procedure support in core machine instructions

Instruction	Intuition	Pseudocode

`CallProcedure` `a n`	Invoke the procedure at code address `a`, creating a new stack frame and passing `n` parameters.	`p_n := pop`
		`...`
		`p_1 := pop`
		`push B`
		`B := length stack - 1`
		`push PC`
		`push p_1`
		`...`
		`push p_n`
		`loadInstruction a`

Table 6.6: New instruction **`CallProcedure`**

Instruction	Intuition	Pseudocode
Return ret	Return from the current procedure invocation, jumping back to the return address, popping the current stack frame, and, depending on ret, possibly returning a value.	BOld := B ra := stack[B + 1] if ret == True then retVal := pop B := stack[B] while length stack > BOld do pop if ret == True then push retVal loadInstruction ra

Table 6.7: New instruction **Return**

To illustrate these additions, consider again in example 6.5 the computation of factorials, this time using a recursive procedure.

```
USING [
    PROCEDURE fac(INT n) RETURNS INT faculty {
        IF n < 0 THEN {
            PRINTI n
            PRINTLNS " is not a natural number!"
            ERROR
        }
        IF n = 0 THEN {
            faculty := 1
        }
        IF n > 0 THEN {
            faculty := n * fac(n - 1)
        }
    }
] DO {
    PRINTS "Please enter a natural number n: "
    INT n
    READ n
    PRINTS "n! = "
    PRINTI fac(n)
}
```

Listing 6.5: Example program `fac1.olang`

The code generator translates program 6.5 to the machine program depicted in listing 6.6.

```
Jump 29                    # skip procedure fac
PushInt 0                  # BEGIN of procedure fac - stack memory allocation for faculty
LoadStack 0                # push n
PushInt 0
CombineBinary Smaller      # IF n < 0
JumpIfFalse 10             # skip IF body if n >= 0
LoadStack 0                # push n
PrintInt                   # PRINTI n
PrintStrLn " is not a natural number!"
Halt                       # ERROR
```

```
10  LoadStack 0                 # push n
11  PushInt 0
12  CombineBinary Equals        # IF n = 0
13  JumpIfFalse 16              # skip IF body if n != 0
14  PushInt 1
15  StoreStack 1                # faculty := 1
16  LoadStack 0                 # push n
17  PushInt 0
18  CombineBinary Greater       # IF n > 0
19  JumpIfFalse 27              # skip IF body if n <= 0
20  LoadStack 0                 # push n
21  LoadStack 0                 # push n
22  PushInt 1
23  CombineBinary Minus         # push n - 1
24  CallProcedure 1 1           # push fac(n - 1)
25  CombineBinary Times         # push n * fac(n - 1)
26  StoreStack 1                # faculty := n * fac(n - 1)
27  LoadStack 1                 # push faculty (return value)
28  Return True                 # END of procedure - return with value of faculty
29  PushInt 0                   # BEGIN of main program - stack memory allocation for n
30  PrintStr "Please enter a natural number n: "
31  PushInt 0
32  StoreStack 0                # initialize n := 0
33  Read
34  StoreStack 0                # READ n
35  PrintStr "n! = "
36  LoadStack 0                 # push n
37  CallProcedure 1 1           # push fac(n)
38  PrintInt                    # PRINTI fac(n)
39  Halt                        # END of main program
```

Listing 6.6: Annotated machine code for example 6.5

Although the machine program is barely longer than the iterative version from section 6.1, the computation is significantly slower due to the required management of stack frames. Tables 6.8 and 6.9 show the trace produced when running the program with $n = 3$.

Step	PC	I	**Stack**	B
0	1	**Jump** 29	[0,0]	0
1	30	**PushInt** 0	[0,0]	0
2	31	**PrintStr** "Please enter a natural number n: "	[0,0,0]	0
3	32	**PushInt** 0	[0,0,0]	0
4	33	**StoreStack** 0	[0,0,0,0]	0
5	34	**Read**	[0,0,0]	0
6	35	**StoreStack** 0	[0,0,0,3]	0
7	36	**PrintStr** "n! = "	[0,0,3]	0
8	37	**LoadStack** 0	[0,0,3]	0
9	38	**CallProcedure** 1 1	[0,0,3,3]	0
10	2	**PushInt** 0	[0,0,3,0,38,3]	3
11	3	**LoadStack** 0	[0,0,3,0,38,3,0]	3
12	4	**PushInt** 0	[0,0,3,0,38,3,0,3]	3
13	5	**CombineBinary Smaller**	[0,0,3,0,38,3,0,3,0]	3
14	6	**JumpIfFalse** 10	[0,0,3,0,38,3,0,0]	3
15	11	**LoadStack** 0	[0,0,3,0,38,3,0]	3
16	12	**PushInt** 0	[0,0,3,0,38,3,0,3]	3
17	13	**CombineBinary Equals**	[0,0,3,0,38,3,0,3,0]	3
18	14	**JumpIfFalse** 16	[0,0,3,0,38,3,0,0]	3
19	17	**LoadStack** 0	[0,0,3,0,38,3,0]	3
20	18	**PushInt** 0	[0,0,3,0,38,3,0,3]	3
21	19	**CombineBinary Greater**	[0,0,3,0,38,3,0,3,0]	3
22	20	**JumpIfFalse** 27	[0,0,3,0,38,3,0,1]	3
23	21	**LoadStack** 0	[0,0,3,0,38,3,0]	3
24	22	**LoadStack** 0	[0,0,3,0,38,3,0,3]	3
25	23	**PushInt** 1	[0,0,3,0,38,3,0,3,3]	3

Table 6.8: Machine trace for program 6.6, $n = 3$, part 1

Step	PC	I	Stack	B
26	24	CombineBinary Minus	[0,0,3,0,38,3,0,3,3,1]	3
27	25	CallProcedure 1 1	[0,0,3,0,38,3,0,3,2]	3
28	2	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2]	8
29	3	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0]	8
30	4	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2]	8
31	5	CombineBinary Smaller	[0,0,3,0,38,3,0,3,3,25,2,0,2,0]	8
32	6	JumpIfFalse 10	[0,0,3,0,38,3,0,3,3,25,2,0,0]	8
33	11	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0]	8
34	12	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2]	8
35	13	CombineBinary Equals	[0,0,3,0,38,3,0,3,3,25,2,0,2,0]	8
36	14	JumpIfFalse 16	[0,0,3,0,38,3,0,3,3,25,2,0,0]	8
37	17	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0]	8
38	18	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2]	8
39	19	CombineBinary Greater	[0,0,3,0,38,3,0,3,3,25,2,0,2,0]	8
40	20	JumpIfFalse 27	[0,0,3,0,38,3,0,3,3,25,2,0,1]	8
41	21	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0]	8
42	22	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2]	8
43	23	PushInt 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,2]	8
44	24	CombineBinary Minus	[0,0,3,0,38,3,0,3,3,25,2,0,2,2,1]	8
45	25	CallProcedure 1 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,1]	8
46	2	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1]	13
47	3	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0]	13
48	4	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1]	13
49	5	CombineBinary Smaller	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,0]	13
50	6	JumpIfFalse 10	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,0]	13
51	11	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0]	13
52	12	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1]	13
53	13	CombineBinary Equals	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,0]	13
54	14	JumpIfFalse 16	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,0]	13
55	17	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0]	13
56	18	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1]	13
57	19	CombineBinary Greater	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,0]	13
58	20	JumpIfFalse 27	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1]	13
59	21	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0]	13
60	22	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1]	13
61	23	PushInt 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,1]	13
62	24	CombineBinary Minus	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,1,1]	13
63	25	CallProcedure 1 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,0]	13
64	2	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0]	18
65	3	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0]	18
66	4	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0,0]	18
67	5	CombineBinary Smaller	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0,0,0]	18
68	6	JumpIfFalse 10	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0,0]	18
69	11	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0]	18
70	12	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0,0]	18
71	13	CombineBinary Equals	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0,0,0]	18
72	14	JumpIfFalse 16	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0,1]	18
73	15	PushInt 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0]	18
74	16	StoreStack 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,0,1]	18
75	17	LoadStack 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,1]	18
76	18	PushInt 0	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,1,0]	18
77	19	CombineBinary Greater	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,1,0,0]	18
78	20	JumpIfFalse 27	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,1,0]	18
79	28	LoadStack 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,1]	18
80	29	Return True	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,13,25,0,1,1]	18
81	26	CombineBinary Times	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1,1]	13
82	27	StoreStack 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,0,1]	13
83	28	LoadStack 1	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,1]	13
84	29	Return True	[0,0,3,0,38,3,0,3,3,25,2,0,2,8,25,1,1,1]	13
85	26	CombineBinary Times	[0,0,3,0,38,3,0,3,3,25,2,0,2,1]	8
86	27	StoreStack 1	[0,0,3,0,38,3,0,3,3,25,2,0,2]	8
87	28	LoadStack 1	[0,0,3,0,38,3,0,3,3,25,2,2]	8
88	29	Return True	[0,0,3,0,38,3,0,3,3,25,2,2,2]	8
89	26	CombineBinary Times	[0,0,3,0,38,3,0,3,2]	3
90	27	StoreStack 1	[0,0,3,0,38,3,0,6]	3
91	28	LoadStack 1	[0,0,3,0,38,3,6]	3
92	29	Return True	[0,0,3,0,38,3,6,6]	3
93	39	PrintInt	[0,0,3,6]	0
94	40	Halt	[0,0,3]	0

Table 6.9: Machine trace for program 6.6, $n = 3$, part 2

6.3 Support for object-oriented features

Supporting O's object-oriented features requires supporting the representation of objects in the abstract machine. As explained in section 3.4, in O, there is a difference between the objects themselves and their addresses held by object variables. It makes sense to keep addresses on the stack, where they will be subject to the stack frame management introduced in section 6.2. For the objects themselves though, this poses a problem. The lifespan of an object is not restricted by the invocation that created it, so it cannot simply be discarded after the end of an invocation. In principle, one could think of ways to try to keep the objects on the stack, too, but this is far from ideal since care must be taken to copy objects that might still be referenced after an invocation ends. Also, the addresses themselves would need to be changed everywhere a reference is kept. The obvious solution for avoiding these problems is keeping the objects in a separate region of memory, which is called the *heap memory*. An object in the abstract machine's heap carries a numeric identifier of the class it belongs to, along with a sequence of data fields that represent the object's fields.

Additionally, the machine must support O's *dynamic binding* mechanism which makes method invocations different from procedure invocations in that the invoked method is dependent on the object at runtime. This is done by introducing *method tables* that hold the addresses of all methods of a given class. Method tables are indexed by numeric method identifiers.

The abstract machine is therefore extended by the following data structures:

- A heap memory `H` that is a map of object addresses to objects,
- an object counter `O` that is used to generate addresses for newly created objects,
- and a set of method tables `MTT` that hold the addresses of all methods, which is effectively a two-dimensional sequence using class and method identifiers as indices.

The instruction cycle is changed to reflect this:

```
code := program
I := code[0]
PC := 1
stack := [0, 0]
B := 0
H := []
O := 0
MTT := []
while I != Halt do executeInstruction
```

Listing 6.7: Instruction cycle for core machine with procedures and object-oriented features

Additionally, five new instructions are required which are displayed in tables 6.10 and 6.11.

Instruction	Intuition	Pseudocode

LoadHeap `i`	Push to the stack the field value of the object with address `a` and field index `i`.	`a := pop`
		`obj := H[a]`
		`push fields(obj)[i]`
		`loadInstruction PC`

Table 6.10: New instructions for support of object-oriented features, part 1

Instruction	Intuition	Pseudocode
StoreHeap i	Store the stack's topmost value to the field with index i, of object with address a where a is the stack's second topmost value.	val := pop a := pop fields(H[a])[i] := ↪ val loadInstruction PC
AllocateHeap n cid	Create on the heap a new object with n fields and class identifier cid and push the object's address to the stack.	H[O] := createObj(n, ↪ cid) push O O := O + 1 loadInstruction PC
CreateMethodTable cid mt	Create a new method table mt = [(id0, a0), ..., (idn, an)] with class identifier cid that holds the code addresses a0 ... an for methods with identifiers id0 ... idn.	MTT[cid] := [(id0, ↪ a0), ..., (idn, ↪ an)] loadInstruction PC
CallMethod mid n	Invokes method with the stack's n + 1 topmost values as parameters, of which the first is the address of the parameter object. The method invoked is the one with index mid in the corresponding method table.	p_n := pop ... p_1 := pop oa := pop push B B := length stack - 1 push PC push oa push p_1 ... push p_n obj := H[oa] cid := classid(obj) loadInstruction ↪ MTT[cid][mid]

Table 6.11: New instructions for support of object-oriented features, part 2

Example 6.8 shows the factorial calculation from 6.2 adapted to use an Intbox-object (see example 3.5) as the accumulator instead of a normal integer.

```
USING [
    CLASS Intbox(INT i)
    FIELDS INT i
    INIT { this.i := i }
    [
        METHOD multiply(INT n) {
            this.i := this.i * n
```

```
        }

        METHOD print() {
            PRINTI this.i
        }
    ]
] DO {
    PRINTS "Please enter a natural number n: "
    INT n
    READ n
    OBJ Intbox faculty
    faculty := Intbox(1)
    IF n < 0 THEN {
        PRINTI n
        PRINTLNS " is not a natural number!"
        ERROR
    }
    WHILE n > 0 DO {
        CALL faculty.multiply(n)
        n := n - 1
    }
    PRINTS "n! = "
    CALL faculty.print()
}
```

Listing 6.8: Example program `fac2.olang`

Example 6.8 translates to the machine code depicted in listing 6.9.

```
Jump 14                             # skip initializer for Intbox
PushInt 0                           # BEGIN of initializer for Intbox - stack memory
 ↪ allocation for 'this'
PushInt (-1)
StoreStack 1                        # initialize this := -1 (null pointer/invalid
 ↪ address)
AllocateHeap 1 0                    # create object of class Intbox
StoreStack 1                        # this := new Intbox
LoadStack 1                         # push this
PushInt 0
StoreHeap 0                         # initialize this.i := 0
LoadStack 1                         # push this
LoadStack 0                         # push i
StoreHeap 0                         # this.i := i
LoadStack 1                         # push this
Return True                         # END of initializer - return with value this
Jump 22                             # skip method multiply
LoadStack 0                         # BEGIN of method multiply - push this
LoadStack 0                         # push this
LoadHeap 0                          # push this.i
LoadStack 1                         # push n
CombineBinary Times                 # push this.i * n
StoreHeap 0                         # this.i := this.i * n
Return False                        # END of method multiply - no return value
Jump 27                             # skip method print
LoadStack 0                         # push this
LoadHeap 0                          # push this.i
PrintInt                            # PRINTI this.i
Return False                        # END of method print - no return value
PushInt 0                           # BEGIN of main program - stack memory allocation
 ↪ for n
PushInt 0                           # stack memory allocation for faculty
CreateMethodTable 0 [(1,23),(0,15)]  # create method table for class Intbox
PrintStr "Please enter a natural number n: "
PushInt 0
StoreStack 0                        # initialize n := 0
```

```
33  Read
34  StoreStack 0                          # READ n
35  PushInt (-1)
36  StoreStack 1                          # initialize faculty := -1
37  PushInt 1
38  CallProcedure 1 1                     # push Intbox(1)
39  StoreStack 1                          # faculty := Intbox(1)
40  LoadStack 0                           # push n
41  PushInt 0
42  CombineBinary Smaller                 # IF n < 0
43  JumpIfFalse 48                        # skip IF body if n >= 0
44  LoadStack 0                           # push n
45  PrintInt                              # PRINTI n
46  PrintStrLn " is not a natural number!"
47  Halt                                  # ERROR
48  LoadStack 0                           # push n
49  PushInt 0
50  CombineBinary Greater                 # WHILE n > 0
51  JumpIfFalse 60                        # skip WHILE body if n <= 0
52  LoadStack 1                           # push faculty
53  LoadStack 0                           # push n
54  CallMethod 0 1                        # push faculty.multiply(n)
55  LoadStack 0                           # push n
56  PushInt 1
57  CombineBinary Minus                   # push n - 1
58  StoreStack 0                          # n := n - 1
59  Jump 48                               # end of WHILE body: return to condition
60  PrintStr "n! = "
61  LoadStack 1                           # push faculty
62  CallMethod 1 0                        # CALL faculty.print()
63  Halt                                  # END of main program
```

Listing 6.9: Annotated machine code for example 6.8

Tables 6.12 and 6.13 show the program trace produced by running example 6.9 with $n = 3$.

Step	PC	I	Stack	B	MTT	H	O
0	1	**Jump** 14	[0,0]	0	[]	[]	0
1	15	**Jump** 22	[0,0]	0	[]	[]	0
2	23	**Jump** 27	[0,0]	0	[]	[]	0
3	28	**PushInt** 0	[0,0]	0	[]	[]	0
4	29	**PushInt** 0	[0,0,0]	0	[]	[]	0
5	30	**CreateMethodTable** 0 [(1,23),(0,15)]	[0,0,0,0]	0	[]	[]	0
6	31	**PrintStr** "Please enter a natural number n: "	[0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[]	0
7	32	**PushInt** 0	[0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[]	0
8	33	**StoreStack** 0	[0,0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[]	0
9	34	**Read**	[0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[]	0
10	35	**StoreStack** 0	[0,0,0,0,3]	0	[(0,[(1,23),(0,15)])]	[]	0
11	36	**PushInt** (-1)	[0,0,3,0]	0	[(0,[(1,23),(0,15)])]	[]	0
12	37	**StoreStack** 1	[0,0,3,0,-1]	0	[(0,[(1,23),(0,15)])]	[]	0
13	38	**PushInt** 1	[0,0,3,-1]	0	[(0,[(1,23),(0,15)])]	[]	0
14	39	**CallProcedure** 1 1	[0,0,3,-1,1]	0	[(0,[(1,23),(0,15)])]	[]	0
15	2	**PushInt** 0	[0,0,3,-1,0,39,1]	4	[(0,[(1,23),(0,15)])]	[]	0
16	3	**PushInt** (-1)	[0,0,3,-1,0,39,1,0]	4	[(0,[(1,23),(0,15)])]	[]	0
17	4	**StoreStack** 1	[0,0,3,-1,0,39,1,0,-1]	4	[(0,[(1,23),(0,15)])]	[]	0
18	5	**AllocateHeap** 1 0	[0,0,3,-1,0,39,1,-1]	4	[(0,[(1,23),(0,15)])]	[]	0
19	6	**StoreStack** 1	[0,0,3,-1,0,39,1,-1,0]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [0])]	1
20	7	**LoadStack** 1	[0,0,3,-1,0,39,1,0]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [0])]	1
21	8	**PushInt** 0	[0,0,3,-1,0,39,1,0,0]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [0])]	1
22	9	**StoreHeap** 0	[0,0,3,-1,0,39,1,0,0,0]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [0])]	1
23	10	**LoadStack** 1	[0,0,3,-1,0,39,1,0]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [0])]	1
24	11	**LoadStack** 0	[0,0,3,-1,0,39,1,0,0]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [0])]	1
25	12	**StoreHeap** 0	[0,0,3,-1,0,39,1,0,0,1]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [0])]	1
26	13	**LoadStack** 1	[0,0,3,-1,0,39,1,0]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
27	14	**Return True**	[0,0,3,-1,0,39,1,0,0]	4	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
28	40	**StoreStack** 1	[0,0,3,-1,0]	0	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
29	41	**LoadStack** 0	[0,0,3,0]	0	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
30	42	**PushInt** 0	[0,0,3,0,3]	0	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
31	43	**CombineBinary Smaller**	[0,0,3,0,3,0]	0	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
32	44	**JumpIfFalse** 48	[0,0,3,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
33	49	**LoadStack** 0	[0,0,3,0]	0	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
34	50	**PushInt** 0	[0,0,3,0,3]	0	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1
35	51	**CombineBinary Greater**	[0,0,3,0,3,0]	0	[(0,[(1,23),(0,15)])]	[(0,**OBJ** 0 [1])]	1

Table 6.12: Machine trace for program 6.9, $n = 3$, part 1

Step	PC	I	Stack	B	MTT	H	O
36	52	JumpIfFalse 60	[0,0,3,0,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
37	53	LoadStack 1	[0,0,3,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
38	54	LoadStack 0	[0,0,3,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
39	55	CallMethod 0 1	[0,0,3,0,0,3]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
40	16	LoadStack 0	[0,0,3,0,0,55,0,3]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
41	17	LoadStack 0	[0,0,3,0,0,55,0,3,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
42	18	LoadHeap 0	[0,0,3,0,0,55,0,3,0,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
43	19	LoadStack 1	[0,0,3,0,0,55,0,3,0,1]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
44	20	CombineBinary Times	[0,0,3,0,0,55,0,3,0,1,3]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
45	21	StoreHeap 0	[0,0,3,0,0,55,0,3,0,3]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [1])]	1
46	22	Return False	[0,0,3,0,0,55,0,3]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
47	56	LoadStack 0	[0,0,3,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
48	57	PushInt 1	[0,0,3,0,3]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
49	58	CombineBinary Minus	[0,0,3,0,3,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
50	59	StoreStack 0	[0,0,3,0,2]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
51	60	Jump 48	[0,0,2,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
52	49	LoadStack 0	[0,0,2,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
53	50	PushInt 0	[0,0,2,0,2]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
54	51	CombineBinary Greater	[0,0,2,0,2,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
55	52	JumpIfFalse 60	[0,0,2,0,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
56	53	LoadStack 1	[0,0,2,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
57	54	LoadStack 0	[0,0,2,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
58	55	CallMethod 0 1	[0,0,2,0,0,2]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
59	16	LoadStack 0	[0,0,2,0,0,55,0,2]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
60	17	LoadStack 0	[0,0,2,0,0,55,0,2,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
61	18	LoadHeap 0	[0,0,2,0,0,55,0,2,0,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
62	19	LoadStack 1	[0,0,2,0,0,55,0,2,0,3]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
63	20	CombineBinary Times	[0,0,2,0,0,55,0,2,0,3,2]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
64	21	StoreHeap 0	[0,0,2,0,0,55,0,2,0,6]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [3])]	1
65	22	Return False	[0,0,2,0,0,55,0,2]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
66	56	LoadStack 0	[0,0,2,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
67	57	PushInt 1	[0,0,2,0,2]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
68	58	CombineBinary Minus	[0,0,2,0,2,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
69	59	StoreStack 0	[0,0,2,0,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
70	60	Jump 48	[0,0,1,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
71	49	LoadStack 0	[0,0,1,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
72	50	PushInt 0	[0,0,1,0,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
73	51	CombineBinary Greater	[0,0,1,0,1,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
74	52	JumpIfFalse 60	[0,0,1,0,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
75	53	LoadStack 1	[0,0,1,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
76	54	LoadStack 0	[0,0,1,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
77	55	CallMethod 0 1	[0,0,1,0,0,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
78	16	LoadStack 0	[0,0,1,0,0,55,0,1]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
79	17	LoadStack 0	[0,0,1,0,0,55,0,1,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
80	18	LoadHeap 0	[0,0,1,0,0,55,0,1,0,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
81	19	LoadStack 1	[0,0,1,0,0,55,0,1,0,6]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
82	20	CombineBinary Times	[0,0,1,0,0,55,0,1,0,6,1]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
83	21	StoreHeap 0	[0,0,1,0,0,55,0,1,0,6]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
84	22	Return False	[0,0,1,0,0,55,0,1]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
85	56	LoadStack 0	[0,0,1,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
86	57	PushInt 1	[0,0,1,0,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
87	58	CombineBinary Minus	[0,0,1,0,1,1]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
88	59	StoreStack 0	[0,0,1,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
89	60	Jump 48	[0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
90	49	LoadStack 0	[0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
91	50	PushInt 0	[0,0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
92	51	CombineBinary Greater	[0,0,0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
93	52	JumpIfFalse 60	[0,0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
94	61	PrintStr "n! = "	[0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
95	62	LoadStack 1	[0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
96	63	CallMethod 1 0	[0,0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
97	24	LoadStack 0	[0,0,0,0,0,63,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
98	25	LoadHeap 0	[0,0,0,0,0,63,0,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
99	26	PrintInt	[0,0,0,0,0,63,0,6]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
100	27	Return False	[0,0,0,0,0,63,0]	4	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1
101	64	Halt	[0,0,0,0]	0	[(0,[(1,23),(0,15)])]	[(0,OBJ 0 [6])]	1

Table 6.13: Machine trace for program 6.9, $n = 3$, part 2

6.4 Notes on the provided implementation

The previous sections present a slightly simplified model of the actual machine from the provided implementation. The provided implementation is based on an `ExceptT`-`State`-monad-transformer which carries in its state a data object that represents the machine. The `ExceptT`-monad is used to implement exception handling - among other reasons, instruction execution can fail due to addresses being out of bounds or required input not being present. A **`Computation`** `a` is then any operation that modifies the machine state and yields some result of type `a`. For example, the `pop`-function is a **`Computation Integer`** - it changes the state of the stack, and yields the popped integer.

CHAPTER 7

Code Generator Implementation for O

On the one side, the parser is able to produce syntax tree representations of programs. On the other side, the abstract machine provides a translation target for the language. The code generator now realizes the translation from syntax trees to machine programs. Like the parser, the code generator is recursive in nature. It is really a set of generators that each generate code for a certain syntactical structure from O. The generators then recursively rely on each other to obtain part solutions to their respective code generation problem. The code generators are covered in section 7.1. At certain points during the code generation, the compatibility of expression types with their surrounding context must be checked. For this, the type of expression is first calculated, and then compared against the requirement of the corresponding "hole" - the context of the expression in the program. The type calculation is outsourced to the *typifiers* that are covered in section 7.2. For more general information about type checking, see [1].

7.1 Code generators

Like the abstract machine, the generators are implemented as `ExceptT`-`State`-transformers - they carry some internal state and can throw exceptions at any point. The exceptions are necessary for providing useful feedback to the end user in case of an erroneous program. This includes errors like references to undefined variables, type errors or ambiguous invocations (see subsection 3.5.2). This chapter will not cover exception handling in detail. The internal state consists of all information that is required to generate the correct code, except for the syntax trees themselves, which are provided as explicit input to the generators. There are four important parts to the internal state: The *prefix length*, the *symbol table*, the *procedure table* and the *class table*. The prefix length denotes the number of preceding machine instructions before the one that is generated next. The symbol table holds a *symbol entry* for each symbol that is currently known. Symbols are introduced either through the declaration of a variable, a formal parameter or a formal return parameter. The only exception to this rule is the symbol `this` that is automatically introduced for the scope of an initializer or method. Similarly, the procedure and class table hold *procedure entries* and *class entries* for each procedure and class that are currently known. Table 7.1 describes the information contained within symbol-, procedure- and class entries.

<table>
<tr><th>Element</th><th>Contained Information</th></tr>
<tr><td>symbol entry</td><td>
<table>
<tr><th>Element</th><th>Description</th></tr>
<tr><td>name</td><td>The string identifier from the O-program</td></tr>
<tr><td>type</td><td>The declared type</td></tr>
<tr><td>position</td><td>The relative position of the symbol's data in any invocation's stack frame that contains it</td></tr>
</table>
</td></tr>
<tr><td>procedure entry</td><td>
<table>
<tr><th>Element</th><th>Description</th></tr>
<tr><td>name</td><td>The name of the procedure in the O-program</td></tr>
<tr><td>parameter types</td><td>The sequence of types of the declared parameters in the formal parameter list of the procedure</td></tr>
<tr><td>return type</td><td>The type of the formal return parameter</td></tr>
<tr><td>address</td><td>The address of the procedure in the machine program</td></tr>
</table>
</td></tr>
<tr><td>class entry</td><td>
<table>
<tr><th>Element</th><th>Description</th></tr>
<tr><td>class identifier</td><td>The numeric identifier assigned to the class</td></tr>
<tr><td>name</td><td>The name of the class in the O-program</td></tr>
<tr><td>(optional) upper class identifier</td><td>If an upper class exists, the numeric identifier of the upper class</td></tr>
<tr><td>field table</td><td>Holds a field entry for each field of the class (see below)</td></tr>
<tr><td>method table</td><td>Holds a method entry for each method of the class (see below)</td></tr>
</table>
</td></tr>
</table>

Table 7.1: Generator state descriptions

Analogous to the symbol table, the field table of a class holds all information about its fields, containing for each field entry a name, type and position. In the case of fields, the position is the index of the field in the object's data segment on the heap. Analogous to the procedure table, the method table of a class holds all information about its methods, with each method entry spanning the information of a procedure entry with an additional numeric *method identifier*.

Some of the elements, such as names and types, are immediately obvious from the program text. Identifiers, addresses and positions on the other hand are calculated during code generation. At the start of code generation, the prefix length is 0, and all tables are empty.

The operations that modify a table or lookup an entry are implemented such that the table is treated like a stack, to realize the variable shadowing introduced in chapter 3.

Since generators modify the internal state, which is implicitly passed into recursive calls, and more importantly, an invoked generator will change the state of the caller as well (as a consequence of keeping a monadic state), there needs to be a convention about state management. In case a generator for procedures modifies for example the symbol table, the symbols need to be cleared from the state afterwards before generation can continue. The convention is the "polluter pays principle" - the generator that causes "pollution" in the state needs to clean it up before terminating.

7.1.1 Programs

A program always consists of a list of classes, procedures and an instruction (the "main program"):

```
data Program =
  Program
    [ClassDeclaration]
    [ProcedureDeclaration]
    Instruction
```

Listing 7.1: Syntax tree data structure for **Program**

The machine code that is generated follows the structure detailed in listing 7.2:

```
<code for classes>
<code for procedures>
# stack memory allocation for main program
# one PushInt instruction for any declared variable
PushInt 0
...
PushInt 0
# create one method table for each class
CreateMethodTable 0 ...
...
CreateMethodTable n ...
<main program instruction code>
Halt
```

Listing 7.2: Machine code layout for **Program**

Listing 7.3 shows a slightly simplified pseudocode version of the actual code generator for programs.

```
generate (Program classes procedures main) =
    # (6) side effects:
    # - populate class table
    # - add initializers to procedure table
    # - increase prefix length
    classInstructions := generate classes
    # (10) side effects:
    # - populate procedure table
    # - increase prefix length
    procedureInstructions := generate procedures
    requiredStackMemory := calculateStackMemoryRequirement main
    stackMemoryAllocationInstructions := requiredStackMemory * [PushInt 0]
    prefixLength += requiredStackMemory
    methodTableInstructions := generateMethodTableInstructions classTable
    prefixLength += length methodTableInstructions
    # generate main instruction
    # side effects are unimportant, since generation ends afterwards
    mainProgramInstructions := generate main
    return classInstructions
            ++ procedureInstructions
            ++ stackMemoryAllocationInstructions
            ++ methodTableInstructions
            ++ mainProgramInstructions
            ++ [Halt]
```

Listing 7.3: Code generator for **Program**

7.1.2 Class declarations

A class declaration consists of a class name, a list of formal parameters for the initializer, optionally the name of the upper class, a list of fields, the initializer code, and a list of method declarations:

```
data ClassDeclaration
  = Class
      ClassName
      FormalParameterList
      (Maybe ClassName)
      [SymbolDeclaration]
      Instruction
      [MethodDeclaration]
```

Listing 7.4: Syntax tree data structure for **ClassDeclaration**

The code generator for classes fulfils three purposes:

1. Adding the class to the class table,
2. generating the initializer code,
3. generating the code for all methods of the class.

The machine code layout is therefore very simple. If the class contains *n* methods:

```
<code for initializer>
<code for method 1>
...
<code for method n>
```

Listing 7.5: Machine code layout for **ClassDeclaration**

The provided implementation of the class generator is approximately equivalent to pseudocode 7.6. The `generateInitializer` helper function utilizes the procedure generator to generate an initializer-procedure for the class. For a class with name `cname`, the initializer-procedure will always have the name `INIT_cname` in the procedure table.

```
generate (Class name parameters mUpperClassName fields initializer methods) =
    # (5) side effects:
    # - add new empty class table entry for 'name' with field information
    # - in case of inheritance, copy relevant information from upper class
    classID := createClassTableEntry name mUpperClassName fields
    # (9) side effects:
    # - add initializer procedure to procedure table
    # - increase prefix length
    initInstructions := generateInitializer name parameters initializer
    # (13) side effects:
    # - add methods to corresponding class table
    # - increase prefix length
    methodInstructions := generateWithContext classID methods
    return initInstructions ++ methodInstructions
```

Listing 7.6: Code generator for **ClassDeclaration**

7.1.3 Method declarations

Due to the underlying similarity with procedure declarations, the syntax tree data structure for method declarations is partly shared with procedure declarations in the form of a **ProcedureHeader**. Both method and procedure declarations therefore contain a name, formal parameter list, an optional return parameter, a list of subprocedures and the code:

```
data MethodDeclaration
    = Method
        ProcedureHeader
        Instruction
data ProcedureHeader
    = ProcedureHeader
        SymbolName
        FormalParameterList
        (Maybe SymbolDeclaration)
        [ProcedureDeclaration]
```

Listing 7.7: Syntax tree data structure for **MethodDeclaration**

The main purpose of the code generator for methods is of course generating code for the method and adding the method to the corresponding class table. If necessary, an inherited method is overridden in the process. Overriding follows the rules introduced in 3.5. If there are subprocedures, their code is also generated. Machine code that is generated for a method with n subprocedures follows the layout depicted in listing 7.8.

```
     Jump END
     <code for subprocedure 1>
     ...
     <code for subprocedure n>
     <code for stack memory allocation>
     <code for initialization of return parameter>
     <method instruction code>
     <return instructions>
END:
```

Listing 7.8: Machine code layout for **MethodDeclaration**

Pseudocode 7.9 describes the **MethodDeclaration**-generator from the provided implementation.

```
generate classID (Method (ProcedureHeader name parameters mReturnParameter
 ↪ subprocedures) code) =
    prefixLength += 1
    # (6) side effects:
    # - add method to method table of class "classID"
    # - in case of inheritance, override inherited method if necessary
    addMethodToClassTable classID name parameters mReturnParameter
    # save the old procedure table for the reset later
    oldProcedureTable := procedureTable
    # (13) side effects:
    # - add subprocedures to procedure table
    # - increase prefix length
    # see procedure generator for details about context
    subProcedureInstructions := generateWithContext NORMAL subprocedures
    # (17) side effects:
    # - add parameters to symbol table
```

```
    # - this includes implicit parameter "this" and return parameter
    thisParam := addMethodParametersToSymbolTable classID parameters mReturnParameter
    # (20) side effect:
    # - increase prefix length
    stackMemoryAllocationInstructions := generateStackMemoryAllocationInstructions
↪   parameters mReturnParameter code
    # (23) side effect:
    # - increase prefix length
    returnParameterInitInstructions := generateMethodReturnParameterInitInstructions
↪   thisParam parameters mReturnParameter
    # (27) side effects:
    # - increase prefix length
    # - modify symbol table (only if code is a single instruction!)
    methodInstructions := generate code
    # (30) side effect:
    # - increase prefix length
    returnInstructions := generateReturnInstructions mReturnParameter
    # reset symbol table
    symbolTable := []
    # cleanup subprocedures from procedure table
    procedureTable := oldProcedureTable

    return [Jump prefixLength]
            ++ subProcedureInstructions
            ++ stackMemoryAllocationInstructions
            ++ returnParameterInitInstructions
            ++ methodInstructions
            ++ returnInstructions
```

Listing 7.9: Code generator for **MethodDeclaration**

7.1.4 Procedure declarations

The syntax tree data structure for procedure declarations is analogous to method declarations:

```
data ProcedureDeclaration
    = Procedure
        ProcedureHeader
        Instruction
data ProcedureHeader
    = ProcedureHeader
        SymbolName
        FormalParameterList
        (Maybe SymbolDeclaration)
        [ProcedureDeclaration]
```

Listing 7.10: Syntax tree data structure for **ProcedureDeclaration**

The code generation for procedure declarations differs from that of method declarations in some ways. For one, the procedure is naturally added to the procedure table instead of a method table. Also, like method declarations, code for a procedure declaration is always generated in a context. The context is either `NORMAL` for normal procedures or `INIT` for initializers. In the case of initializers, additional instructions are generated to allocate a new object on the heap and initialize all fields to a default value. Machine code that is generated for a procedure with n subprocedures follows the layout depicted in listing 7.11.

```
Jump END
<code for subprocedure 1>
```

```
    ...
    <code for subprocedure n>
    <code for stack memory allocation>
    <code for initialization of return parameter>
    <IF INIT-procedure: code for heap memory allocation>
    <procedure instruction machine code>
    <return instructions>
END:
```

Listing 7.11: Machine code layout for **ProcedureDeclaration**

Pseudocode 7.12 describes the **ProcedureDeclaration**-generator from the provided implementation.

```
generate kind (Procedure (ProcedureHeader name parameters mReturnParameter subprocedures) code) =
    prefixLength += 1
    # (5) side effect:
    # - add procedure to procedure table
    addToProcedureTable name parameters mReturnParameter
    # save the old procedure table for the reset later
    oldProcedureTable := procedureTable
    # (11) side effects:
    # - add subprocedures to procedure table
    # - increase prefix length
    subProcedureInstructions := generateWithContext NORMAL subprocedures
    # (15) side effect:
    # - add parameters to symbol table
    # - this includes return parameter
    addProcedureParametersToSymbolTable parameters mReturnParameter
    # (18) side effect:
    # - increase prefix length
    stackMemoryAllocationInstructions := generateStackMemoryAllocationInstructions parameters mReturnParameter code
    # (21) side effect:
    # - increase prefix length
    returnParameterInitInstructions := generateProcedureReturnParameterInitInstructions parameters mReturnParameter
    # (24) side effect:
    # - increase prefix length
    heapMemoryAllocationInstructions := generateHeapMemoryAllocationInstructions kind mReturnParameter
    # (28) side effects:
    # - increase prefix length
    # - modify symbol table (only if code is a single instruction!)
    procedureInstructions := generate code
    # (31) side effect:
    # - increase prefix length
    returnInstructions := generateReturnInstructions mReturnParameter
    # reset symbol table
    symbolTable := []
    # cleanup subprocedures from procedure table
    procedureTable := oldProcedureTable

    return [Jump prefixLength]
            ++ subProcedureInstructions
            ++ stackMemoryAllocationInstructions
            ++ returnParameterInitInstructions
            ++ heapMemoryAllocationInstructions
            ++ procedureInstructions
            ++ returnInstructions
```

Listing 7.12: Code generator for **ProcedureDeclaration**

7.1.5 Instructions

For every possible instruction (see 3.1 and 3.2), there is one alternative in the syntax tree data type.

```
data Instruction
    = Assignment SymbolReference Expression
    | SymbolDeclarationInstruction SymbolDeclaration
    | CallInstruction Call
    | Read SymbolName
    | PrintI Expression
    | PrintS String
    | PrintLnS String
    | Error
    | Block (NonEmpty Instruction)
    | IfThen Condition Instruction
    | While Condition Instruction
data SymbolReference
    = NameReference SymbolName
    | FieldReference SymbolName SymbolName
data SymbolDeclaration
    = IntDeclaration IntSymbolDeclaration
    | ObjectDeclaration ObjectSymbolDeclaration
```

Listing 7.13: Syntax tree data structure for `Instruction`

Because of this, there is no single machine code layout for all cases. Every case needs to be treated independently - for `SymbolReference` and `SymbolDeclaration`, there needs to be an additional distinction between the kind of reference or declaration.

7.1.5.1 Basic instructions

For a simple variable assignment, first the code for the expression and then the store instruction is generated:

```
<code for expression>
StoreStack ...
```

Listing 7.14: Machine code layout for `Instruction`, variable assignment

The code additionally checks that the expression and variable types are compatible according to the subtyping rules:

```
generate (Assignment (NameReference name) expr) =
    (symPos, symType) := lookupSymbolPosAndTypeByName name
    exprType := typify expr
    checkTypeCompatibility exprType symType
    # (7) side effect:
    # - increase prefix length
    exprInstructions := generate expr
    prefixLength += 1
    return exprInstructions
            ++ [StoreStack symPos]
```

Listing 7.15: Code generator for `Instruction`, variable assignment

For a field assignment, before the expression code, there is an additional load instruction to load the object address:

```
LoadStack ...
<code for expression>
StoreHeap ...
```

Listing 7.16: Machine code layout for **Instruction**, field assignment

```
generate (Assignment (FieldReference obj field) expr) =
    (objPos, objType) := lookupSymbolPosAndTypeByName obj
    (fieldPos, fieldType) := lookupFieldPosAndTypeByTypeAndFieldName objType field
    exprType := typify expr
    checkTypeCompatibility exprType fieldType
    prefixLength += 1
    # (9) side effect:
    # - increase prefix length
    exprInstructions := generate expr
    prefixLength += 1
    return [LoadStack objPos]
            ++ exprInstructions
            ++ [StoreHeap fieldPos]
```

Listing 7.17: Code generator for **Instruction**, field assignment

An integer variable declaration is translated by adding it to the symbol table and storing the default value 0 to its position in the stack frame:

```
generate (SymbolDeclarationInstruction (IntDeclaration (Int n))) =
    # (4) side effect:
    # - add new symbol to symbol table
    pos := addSymbolToTable n INT
    prefixLength += 2
    return [PushInt 0, StoreStack pos]
```

Listing 7.18: Code generator for **Instruction**, integer variable declaration

The translation of object variable declaration is analogous, with an additional check for class validity, and a different default value of -1 (the invalid address):

```
generate (SymbolDeclarationInstruction (ObjectDeclaration (Object cname name))) =
    checkClassValidity cname
    # (5) side effect:
    # - add new symbol to symbol table
    pos := addSymbolToTable name (OBJ cname)
    prefixLength += 2
    return [PushInt (-1), StoreStack pos]
```

Listing 7.19: Code generator for **Instruction**, object variable declaration

The translation of a `CALL`-instruction is just the translation of the **Call** that is carried within, but since no assignment is performed, there is an additional check to make sure the type is empty (see section 7.2). This ensures that no unconsumed values remain on the stack after evaluation.

```
generate (CallInstruction call) =
    t <- typify call
    case t of
      # (6) side effect:
      # - increase prefix length
      Nothing -> return (generate call)
      Just _ -> error ...
```

Listing 7.20: Code generator for **Instruction**, call

A READ-instruction is translated similarly to an assignment to an integer variable, with a **Read** instead of expression code:

```
generate (SyntaxTree.Read name) =
    (pos, t) := lookupSymbolPosAndTypeByName name
    checkTypeCompatibility INT t
    prefixLength += 2
    return [MachineInstruction.Read, StoreStack pos]
```

Listing 7.21: Code generator for **Instruction**, read

A PRINTI-instruction is also translated similarly to an assignment to an integer variable, but the result is printed instead of stored:

```
generate (PrintI expr) =
    t := typify expr
    checkTypeCompatibility t INT
    # (6) side effect:
    # - increase prefix length
    exprInstructions := generate expr
    prefixLength += 1
    return exprInstructions ++ [PrintInt]
```

Listing 7.22: Code generator for **Instruction**, integer print

PRINTS and PRINTSLN are directly translated to **PrintStr** and **PrintStrLn** machine instructions, respectively:

```
generate (PrintS msg) =
    prefixLength += 1
    return [PrintStr msg]
generate (PrintLnS msg) =
    prefixLength += 1
    return [PrintStrLn msg]
```

Listing 7.23: Code generator for **Instruction**, string print

And finally, the ERROR instruction is directly translated to the machine instruction **Halt**:

```
generate Error =
    prefixLength += 1
    return [Halt]
```

Listing 7.24: Code generator for **Instruction**, error

7.1.5.2 Composite instructions

An instruction block is simply translated by translating all individual instructions in order, resetting the symbol table afterwards to implement scoping:

```
generate (Block oInstructions) =
    oldSymbolTable := symbolTable
    # (6) side effects:
    # - increase prefix length
    # - add new symbols to symbol table
    mInstructions := generate oInstructions
    symbolTable := oldSymbolTable
    return mInstructions
```

Listing 7.25: Code generator for **Instruction**, block

An IF-THEN-conditional is translated by first generating code for the condition, then a conditional jump, followed by the code for the body of the conditional:

```
    <code for condition>
    JumpIfFalse END
    <code for body>
END:
```

Listing 7.26: Machine code layout for **Instruction**, IF-THEN-conditional

As with instruction blocks, the symbol table is reset after generating the body:

```
generate (IfThen cond body) =
    # (4) side effect:
    # - increase prefix length
    condInstructions := generate cond
    prefixLength += 1
    oldSymbolTable := symbolTable
    # (10) side effects:
    # - increase prefix length
    # - add new symbols to symbol table
    bodyInstructions := generate body
    symbolTable := oldSymbolTable
    return condInstructions
            ++ [JumpIfFalse prefixLength]
            ++ bodyInstructions
```

Listing 7.27: Code generator for **Instruction**, IF-THEN-conditional

WHILE-loops are the most complicated to translate. There are multiple valid translations that can be used, but one of the arguably most simple ones uses two jumps, where one is conditional and one is not. The translation is analogous to IF-THEN-conditionals, with the body getting an additional **Jump** instruction at the end to go back to the condition:

```
START: <code for condition>
       JumpIfFalse END
       <code for body>
       Jump START
  END:
```

Listing 7.28: Machine code layout for **Instruction**, WHILE-loop

Again, the symbol table needs to be reset after translating the body:

```
generate (While cond body) =
    oldSymbolTable := symbolTable
    start := prefixLength
    # (6) side effect:
    # - increase prefix length
    condInstructions := generate cond
    prefixLength += 1
    # (11) side effects:
    # - increase prefix length
    # - add new symbols to symbol table
    bodyInstructions := generate body
    prefixLength += 1
    symbolTable := oldSymbolTable
    end := prefixLength
    return condInstructions
            ++ [JumpIfFalse end]
            ++ bodyInstructions
            ++ [Jump start]
```

Listing 7.29: Code generator for **Instruction**, WHILE-loop

7.1.6 Calls

Since the syntax tree data structure for calls in O represents not only procedure and method calls, but also simple references to variables and object fields, the data type has four alternatives for each case:

```
data Call
    = SymbolReference SymbolReference
    | Call SymbolReference ActualParameterList
```

Listing 7.30: Syntax tree data structure for **Call**

The simplest case is a variable reference. The position in the stack frame is looked up, and a machine instruction is generated to push the value onto the stack:

```
generate (SymbolReference (NameReference name)) =
    pos := lookupSymbolPosByName name
    prefixLength += 1
    return [LoadStack pos]
```

Listing 7.31: Code generator for **Call**, symbol reference

For a field reference, the address of the object is first pushed onto the stack. Then, the field's value is pushed onto the stack using the object address and field position:

```
generate (SymbolReference (FieldReference obj field)) =
    (objPos, t) := lookupSymbolPosAndTypeByName obj
    fieldPos := lookupFieldPosByTypeAndFieldName t field
    prefixLength += 2
    return [LoadStack objPos, LoadHeap fieldPos]
```

Listing 7.32: Code generator for **Call**, field reference

For a procedure call with address a and n parameter expressions, the machine code layout is depicted in listing 7.33.

```
<code for expression 1>
...
<code for expression n>
CallProcedure a n
```

Listing 7.33: Machine code layout for **Call**, procedure invocation

First, the types of all expressions in the actual parameter list are calculated. This type information is used to calculate the given invocation's matching set minimum (see definition 3.5.2) and obtain the corresponding address. After generating code for all parameter expressions, the **CallProcedure**-instruction is appended:

```
generate (Call (NameReference name) actualParameterList) =
    paramTypes := typify actualParameterList
    procAddress := calculateMatchingSetMinimumAddressForProcedureInvocation name
 ↪  paramTypes
    # (6) side effect:
    # - increase prefix length
    paramInstructions := generate actualParameterList
```

```
    prefixLength += 1
    return paramInstructions
            ++ [CallProcedure procAddress (length actualParameterList)]
```

Listing 7.34: Code generator for **Call**, procedure invocation

For a method call with method identifier *id* and *n* parameter expressions, the machine code layout is analogous to procedure calls, with an additional load instruction at the start to load the object address. It is depicted in listing 7.35.

```
LoadStack ...
<code for expression 1>
...
<code for expression n>
CallMethod id n
```

Listing 7.35: Machine code layout for **Call**, method invocation

Keeping these differences in mind, the generator code is analogous to the procedure invocation generator:

```
generate (Call (FieldReference objName methodName) actualParameterList) =
    paramTypes := typify actualParameterList
    methodID := calculateMatchingSetMinimumIDForMethodInvocation objName methodName
 ↪  paramTypes
    objPos := lookupSymbolPosByName objName
    prefixLength += 1
    # (8) side effect:
    # - increase prefix length
    paramInstructions := generate actualParameterList
    prefixLength += 1
    return [LoadStack objPos]
            ++ concat paramInstructions
            ++ [CallMethod methodID (length actualParameterList)]
```

Listing 7.36: Code generator for **Call**, method invocation

7.1.7 Conditions

A condition in O is either a comparison of two expressions or a negation of another condition:

```
data Condition
    = Comparison Expression Relation Expression
    | Negation Condition
```

Listing 7.37: Syntax tree data structure for **Condition**

In case of a comparison, the generator needs to first generate the instructions for the first expression, then the second. After evaluation, both argument values are stored on top of the stack - so they can be combined according to the relation:

```
<code for left expression>
<code for right expression>
CombineBinary ...
```

Listing 7.38: Machine code layout for **Condition**, comparison

The code performs additional checks to ensure that both expression have an integral type:

```
generate (Comparison left relation right) =
    tLeft := typify left
    tRight := typify right
    checkTypeCompatibility tLeft INT
    checkTypeCompatibility tRight INT
    # (8) side effect:
    # - increase prefix length
    leftInstructions := generate left
    # (11) side effect:
    # - increase prefix length
    rightInstructions := generate right
    prefixLength += 1
    return leftInstructions
            ++ rightInstructions
            ++ [CombineBinary relation]
```

Listing 7.39: Code generator for **Condition**, comparison

The case for negations is very simple. Since comparisons are always of type `BOOL` (if the generator terminates successfully), no type check has to be performed. The translation consists of simply appending a **CombineUnary Not** machine instruction to the code of the inner condition.

```
generate (Negation cond) =
    # (4) side effect:
    # - increase prefix length
    condInstructions := generator cond
    prefixLength += 1
    return condInstructions ++ [CombineUnary Not]
```

Listing 7.40: Code generator for **Condition**, negation

7.1.8 Expressions

An expression carries a non-empty list of terms that each carry a sign:

```
data Expression = Expression (NonEmpty (Sign, Term))
data Sign = Plus | Minus
```

Listing 7.41: Syntax tree data structure for **Expression**

Note that this simple representation comes with the small drawback that a singular term must carry a sign with it in the syntax tree, even it turns out to be of an object type. The parser handles this by appending a plus-sign automatically if no sign is present. This leads to the small but harmless grammatical problem that for any expression `expr` without a preceding sign, a plus sign can be prepended without changing the meaning - even if it is not an arithmetic expression.

That being said, the machine code layout for expressions is simple:

```
<code for first term>
<code for term 2>
CombinaBinary <sign 2>
...
<code for term n>
CombineBinary <sign n>
```

Listing 7.42: Machine code layout for **Expression**

The first term needs to be treated differently for the rule to work. If the first term carries a plus sign, then the machine code is just the code of the term - this works even if the term has an object type. If the first term carries a minus sign, the value of the first term must be negated:

```
generate (Expression ((sign, term) :| signTerms)) =
    firstTermInstructions := case sign of
      Plus -> generate term
      Minus -> do
        prefixLength += 1
        # (8) side effect:
        # - increase prefix length
        termInstructions := generate term
        prefixLength += 1
        return [PushInt 0]
                ++ termInstructions
                ++ [CombineBinary Minus]
    # (15) side effect:
    # - increase prefix length
    signTermsInstructions := generate signTerms
    return firstTermInstructions ++ signTermsInstructions
    where
      generate (sign', term') =
        # (21) side effect:
        # - increase prefix length
        termInstructions := generate term'
        prefixLength += 1
        return termInstructions ++ [CombineBinary sign']
```

Listing 7.43: Code generator for **Expression**

Since type checks are already performed on an expression before the generator is invoked, the generator for expressions does not need to perform any further type checks by itself.

7.1.9 Terms

Analogous to expressions, a term is a non-empty list of factors, where from the second factor onward, every factor carries an operator that is either $*$ or $/$:

```
data Term = Term Factor [(Operator, Factor)]
data Operator = Times | Divide
```

Listing 7.44: Syntax tree data structure for **Term**

Again, the first factor is treated differently. The first factor is generated directly, while the other ones each get the corresponding operation appended to their code:

```
<code for first factor>
<code for factor 2>
CombinaBinary <operator 2>
...
<code for factor n>
CombineBinary <operator n>
```

Listing 7.45: Machine code layout for **Term**

Like the generator for expressions, the term generator does no type checking. This does not harm the type safety of O, since typification of expressions is established recursively including terms and factors (see section 7.2).

```
generate (Term factor operatorFactors) =
    # (4) side effect:
    # - increase prefix length
    firstFactorInstructions := generate factor
    # (7) side effect:
    # - increase prefix length
    otherFactorsInstructions := generate operatorFactors
    return firstFactorInstructions ++ otherFactorsInstructions
    where
      generate (op, factor') =
        # (13) side effect:
        # - increase prefix length
        factor'Instructions := generate factor'
        prefixLength += 1
        return factor'Instructions ++ [CombineBinary op]
```

Listing 7.46: Code generator for **Term**

7.1.10 Factors

A factor can be either a **Call**, a class instantiation, an integer or a composite factor carrying an expression inside:

```
data Factor
    = CallFactor Call
    | ClassInstantiation ClassName ActualParameterList
    | Integer Integer
    | CompositeFactor Expression
```

Listing 7.47: Syntax tree data structure for **Factor**

The cases are all so simple that they barely require explanation. Again, the type-check is already established on function call, so additional checks are not required.

```
generate (CallFactor call) = generate call
# for class name 'cname', generateInitializer generates the initializer as a procedure
 ↪ with name "INIT_cname"
generate (ClassInstantiation cname actualParameterList) = generate (Call (NameReference
 ↪ ("INIT_" ++ cname)) actualParameterList)
generate (Integer n) =
  prefixLength += 1
  return [PushInt n]
generate (CompositeFactor expr) = generate expr
```

Listing 7.48: Code generator for **Factor**

7.2 Type checking

The provided implementation does type checking in two parts. Code generators for instructions and calls use typifiers to calculate types and check the compatibility of the resulting type with the given context. The typifiers only implement the type calculations themselves. Therefore, they do not generate any machine code or modify the state of the code generator. Procedure and method invocations can yield nothing, which typifiers account for by not just returning a type, but an indicator for if the given typified element yields a value at all. The provided implementation accounts for this by defining

```
data OptionalType = Maybe Type
```

The result of typification is then either **`Nothing`** or **`Just`** `someType`. This section will present the typifiers from the provided implementation not as code, but as logical rules of inference. In the following, the set of symbol-, procedure and class tables will just be represented as Γ. The notation $e :: \tau$ reads "e is of type τ". The following typing rules cover statements of the form $\Gamma \vdash \phi$, reading "in the context of Γ, it can be deduced that ϕ holds". All rules have some number of premises and a conclusion. A rule allows to deduce the conclusion if all premises hold. There are rules with zero premises which are called *axioms*. The conclusion of an axiom always holds. The rules are of the form

$$\frac{premise_1 \ \ldots \ premise_n}{conclusion}\,\text{Name}$$

They can be applied recursively until all premises can be deduced true (by applying axioms).

7.2.1 Calls

The rules CALL1 and CALL2 represent lookups in the symbol- and class tables. They allow the deduction for a variable or field reference if the type can be looked up from the tables.

$$\frac{}{\Gamma \cup \{\texttt{n} :: \tau\} \vdash \texttt{SymbolReference (NameReference n)} :: \tau}\,\text{CALL1 (axiom)}$$

$$\frac{}{\Gamma \cup \{\texttt{o.f} :: \tau\} \vdash \texttt{SymbolReference (FieldReference o f)} :: \tau}\,\text{CALL2 (axiom)}$$

The CALL3 and CALL4 rule are the most complicated rules. For procedure and method invocations respectively, they allow the deduction of the type of the return parameter of the invoked procedure or method only if it corresponds to the minimum of the matching set (see subsection 3.5.2).

$$\frac{\Gamma \vdash e_1 :: \sigma_1 \quad \ldots \quad \Gamma \vdash e_n :: \sigma_n \quad (\pi_1, \ldots, \pi_n) = min(M(\Gamma, \texttt{p}(\sigma_1, \ldots, \sigma_n)))}{\Gamma = \Gamma' \cup \{\texttt{p}(\pi_1, \ldots, \pi_n) :: \tau\} \vdash \texttt{Call (NameReference p)}\,[e_1, \ldots, e_n] :: \tau}\,\text{CALL3}$$

$$\frac{\Gamma \vdash e_1 :: \sigma_1 \quad \ldots \quad \Gamma \vdash e_n :: \sigma_n \quad (\pi_1, \ldots, \pi_n) = min(M(\Gamma, \texttt{o.m}(\sigma_1, \ldots, \sigma_n)))}{\Gamma = \Gamma' \cup \{\texttt{o.m}(\pi_1, \ldots, \pi_n) :: \tau\} \vdash \texttt{Call (FieldReference o m)}\,[e_1, \ldots, e_n] :: \tau}\,\text{CALL4}$$

Note that despite the notation suggesting otherwise, the list of expressions for CALL3 and CALL4 can actually be empty, leaving the equality as the only premise of the rule in that instance.

7.2.2 Expressions

An expression of only one positive term has the type of the term. As noted in section 7.1, this is to include terms that turn out to be of object type:

$$\frac{\Gamma \vdash t :: \tau}{\Gamma \vdash \texttt{Expression}\,[(+, t)] :: \tau}\,\text{EXPR1}$$

An expression with one or more terms can be deduced to be of type `INT` if the terms are all of the type `INT`:

$$\frac{\Gamma \vdash t_1 :: \texttt{Just INT} \quad \ldots \quad \Gamma \vdash t_n :: \texttt{Just INT} \quad n \geq 1}{\Gamma \vdash \texttt{Expression}\,[(s_1, t_1), \ldots, (s_n, t_n)] :: \texttt{Just INT}}\,\text{EXPR2}$$

There is an overlap of rules EXPR1 and EXPR2 for the case of positive singular terms of type `INT`. The right decision is to pick EXPR1 if possible, since it allows for the deduction of both object types and `INT`.

7.2.3 Terms

The rules for terms similarly allow the deduction of object types, but without overlapping:

$$\frac{\Gamma \vdash f :: \tau}{\Gamma \vdash \texttt{Term}\ f\ [] :: \tau}\ \text{TERM1}$$

$$\frac{\Gamma \vdash f_1 :: \texttt{Just INT} \quad \ldots \quad \Gamma \vdash f_n :: \texttt{Just INT} \quad n \geq 2}{\Gamma \vdash \texttt{Term}\ f_1\ [(op_2, f_2), \ldots, (op_n, f_n)] :: \texttt{Just INT}}\ \text{TERM2}$$

7.2.4 Factors

The rules for factors are simple. Since a call factor just represents the call it carries, it inherits its type:

$$\frac{\Gamma \vdash c :: \tau}{\Gamma \vdash \texttt{CallFactor}\ c :: \tau}\ \text{FAC1}$$

A class instantiation is really a procedure invocation of an initializer-procedure, which is reflected in its typing rule:

$$\frac{\Gamma \vdash \texttt{Call (NameReference (INIT\_ ++ cn))}\ [e_1, \ldots, e_n] :: \tau}{\Gamma \vdash \texttt{ClassInstantiation cn}\ [e_1, \ldots, e_n] :: \tau}\ \text{FAC2}$$

Of course, independently of Γ, an `Integer` n is always integral:

$$\frac{}{\Gamma \vdash \texttt{Integer}\ n :: \texttt{Just INT}}\ \text{FAC3 (axiom)}$$

Lastly, in similar fashion to FAC1, a composite factor inherits the type of its expression:

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash \texttt{CompositeFactor}\ e :: \tau}\ \text{FAC4}$$

7.2.5 Deduction algorithm

The rules are deterministic with exception to the ambiguity between EXPR1 and EXPR2. In case of ambiguity, EXPR1 should be picked. Using this, deduction can be carried out by repeatedly applying the best matching rule to the emerging premises, while leaving the deduced type as a placeholder. This builds a tree where the conclusions and premises are the nodes, with the rule instantiations connecting them. If the syntactical element in question is well-typed, all leaves of the tree are either a solvable equation involving the calculation of the matching set (see CALL3 and CALL4) or an invocation of an axiom (see CALL1, CALL2 and FAC3). The type of the syntactical element can then be deduced top-down, starting from the leaves. This mimics the evaluation of the typifiers from the provided implementation.

7.3 Example

To illustrate the inner workings of the code generator, one might consider example 6.8 again. Tables 7.2 to 7.7 show snapshots at certain important points during the code generation. At each point, the relevant O-program code is provided, showing the corresponding machine code, and keeping the comments from 6.9. In the right column of each table, the code generator state after translating the displayed code is shown.

O-program code	Machine code	State after
1 `USING [`	None	**Symbol Table**: Empty **Procedure Table**: Empty **Class Table**: Empty

Table 7.2: Code generation for example program `fac2.olang`, part 1

	O-program code		Machine code	State after
2	`CLASS Intbox(INT i)`	0	`Jump 14 # skip initializer for Intbox`	**Procedure Table**: **Name**: `INIT_Intbox`; **Parameter Types**: `[INT]`; **Return Type**: `Just OBJ Intbox`; **Address**: 1 **Class Table**: **ID**: 0; **Name**: `Intbox`; **Upper Class ID**: None **Field Table for `Intbox`**: **Name**: `i`; **Type**: `INT`; **Position**: 0 **Method Table for `Intbox`**: Empty **Symbol Table**: Empty
3	`FIELDS INT i`			
4	`INIT { this.i := i }`			
5	`[`	1	`PushInt 0 # BEGIN of initializer for Intbox - stack memory allocation for 'this'`	
		2	`PushInt (-1)`	
		3	`StoreStack 1 # initialize this := -1 (null pointer/invalid address)`	
		4	`AllocateHeap 1 0 # create object of class Intbox`	
		5	`StoreStack 1 # this := new Intbox`	
		6	`LoadStack 1 # push this`	
		7	`PushInt 0`	
		8	`StoreHeap 0 # initialize this.i := 0`	
		9	`LoadStack 1 # push this`	
		10	`LoadStack 0 # push i`	
		11	`StoreHeap 0 # this.i := i`	
		12	`LoadStack 1 # push this`	
		13	`Return True # END of initializer - return with value this`	

Table 7.3: Code generation for example program `fac2.olang`, part 2

O-program code

```
6  METHOD multiply(INT
   ↪  n) {
7      this.i := this.i
   ↪  * n
8  }
```

Machine code

```
14 Jump 22 # skip
   ↪  method multiply
15 LoadStack 0 # BEGIN
   ↪  of method
   ↪  multiply - push
   ↪  this
16 LoadStack 0 # push
   ↪  this
17 LoadHeap 0 # push
   ↪  this.i
18 LoadStack 1 # push n
19 CombineBinary Times
   ↪  # push this.i *
   ↪  n
20 StoreHeap 0 # this.i
   ↪  := this.i * n
21 Return False # END
   ↪  of method
   ↪  multiply - no
   ↪  return value
```

State after

Procedure Table

Name	Parameter Types	Return Type	Address
INIT_Intbox	[INT]	Just OBJ Intbox	1

Class Table

ID	Name	Upper Class ID
0	Intbox	None

Field Table for Intbox

Name	Type	Position
i	INT	0

Method Table for Intbox

ID	Name	Parameter Types	Return Type	Address
0	multiply	[INT]	Nothing	15

Symbol Table

Empty

Table 7.4: Code generation for example program `fac2.olang`, part 3

O-program code

```
10 METHOD print() {
11     PRINTI this.i
12 }
13 ] DO {
```

Machine code

```
22 Jump 27 # skip
   ↪  method print
23 LoadStack 0 # push
   ↪  this
24 LoadHeap 0 # push
   ↪  this.i
25 PrintInt # PRINTI
   ↪  this.i
26 Return False # END
   ↪  of method print
   ↪  - no return
   ↪  value
```

State after

Procedure Table

Name	Parameter Types	Return Type	Address
INIT_Intbox	[INT]	Just OBJ Intbox	1

Class Table

ID	Name	Upper Class ID
0	Intbox	None

Field Table for Intbox

Name	Type	Position
i	INT	0

Method Table for Intbox

ID	Name	Parameter Types	Return Type	Address
0	multiply	[INT]	Nothing	15
1	print	[]	Nothing	23

Symbol Table

Empty

Table 7.5: Code generation for example program `fac2.olang`, part 4

O-program code

```
15 PRINTS "Please enter a natural number n: "
16 INT n
17 READ n
18 OBJ Intbox faculty
19 faculty := Intbox(1)
20 IF n < 0 THEN {
21     PRINTI n
22     PRINTLNS " is not a natural number!"
23     ERROR
24 }
25 WHILE n > 0 DO {
26     CALL faculty.multiply(n)
27     n := n - 1
28 }
29 PRINTS "n! = "
30 CALL faculty.print()
```

Machine code

```
27 PushInt 0 # BEGIN of main program - stack memory allocation for n
28 PushInt 0 # stack memory allocation for faculty
29 CreateMethodTable 0 [(1,23),(0,15)] # create method table for class Intbox
30 PrintStr "Please enter a natural number n: "
31 PushInt 0
32 StoreStack 0 # initialize n := 0
33 Read
34 StoreStack 0 # READ n
35 PushInt (-1)
36 StoreStack 1 # initialize faculty := -1
37 PushInt 1
38 CallProcedure 1 1 # push Intbox(1)
39 StoreStack 1 # faculty := Intbox(1)
40 LoadStack 0 # push n
41 PushInt 0
42 CombineBinary Smaller # IF n < 0
43 JumpIfFalse 48 # skip IF body if n >= 0
44 LoadStack 0 # push n
45 PrintInt # PRINTI n
46 PrintStrLn " is not a natural number!"
47 Halt # ERROR
48 LoadStack 0 # push n
49 PushInt 0
50 CombineBinary Greater # WHILE n > 0
51 JumpIfFalse 60 # skip WHILE body if n <= 0
52 LoadStack 1 # push faculty
53 LoadStack 0 # push n
54 CallMethod 0 1 # push faculty.multiply(n)
55 LoadStack 0 # push n
56 PushInt 1
57 CombineBinary Minus # push n - 1
58 StoreStack 0 # n := n - 1
59 Jump 48 # end of WHILE body: return to condition
60 PrintStr "n! = "
61 LoadStack 1 # push faculty
62 CallMethod 1 0 # CALL faculty.print()
```

State after

Procedure Table

Name	Parameter Types	Return Type	Address
INIT_Intbox	[INT]	Just OBJ Intbox	1

Class Table

ID	Name	Upper Class ID
0	Intbox	None

Field Table for Intbox

Name	Type	Position
i	INT	0

Method Table for Intbox

ID	Name	Parameter Types	Return Type	Address
0	multiply	[INT]	Nothing	15
1	print	[]	Nothing	23

Symbol Table

Name	Type	Position
n	INT	0
faculty	OBJ Intbox	1

Table 7.6: Code generation for example program `fac2.olang`, part 5

O-program code	Machine code	State after
31 `}` 63	`Halt # END of main program`	

Procedure Table

Name	Parameter Types	Return Type	Address
`INIT_Intbox`	`[INT]`	`Just OBJ Intbox`	1

Class Table

ID	Name	Upper Class ID
0	`Intbox`	None

Field Table for `Intbox`

Name	Type	Position
`i`	`INT`	0

Method Table for `Intbox`

ID	Name	Parameter Types	Return Type	Address
0	`multiply`	`[INT]`	`Nothing`	15
1	`print`	`[]`	`Nothing`	23

Symbol Table

Empty

Table 7.7: Code generation for example program `fac2.olang`, part 6

Lastly, Figure 7.1, Figure 7.2 and Figure 7.3 show the type derivations for the expressions in lines 19, 26 and 30 from example program 6.8. They all assume Γ to represent the generator state described in Table 7.6.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{Integer 1} :: \texttt{Just INT}}\,\text{FAC3}}{\Gamma \vdash \texttt{Term (Integer 1) []} :: \texttt{Just INT}}\,\text{TERM1}}{\Gamma \vdash \texttt{Expression [(+,Term (Integer 1) [])]} :: \texttt{Just INT}}\,\text{EXPR1} \quad (\texttt{Just INT}) = min(M(\Gamma, \texttt{INIT\_Intbox}(\texttt{Just INT})))}{\Gamma \vdash \texttt{Call (NameReference INIT\_Intbox) [Expression [(+,Term (Integer 1) [])]]} :: \texttt{Just OBJ Intbox}}\,\text{CALL3}}{\Gamma \vdash \texttt{ClassInstantiation Intbox [Expression [(+,Term (Integer 1) [])]]} :: \texttt{Just OBJ Intbox}}\,\text{FAC2}$$

Figure 7.1: Type derivation for `Intbox(1)`, `fac2.olang`, line 19

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{SymbolReference (NameReference n)} :: \texttt{Just INT}}\,\text{CALL1}}{\Gamma \vdash \texttt{CallFactor (SymbolReference (NameReference n))} :: \texttt{Just INT}}\,\text{FAC1}}{\Gamma \vdash \texttt{Term (CallFactor (SymbolReference (NameReference n))) []} :: \texttt{Just INT}}\,\text{TERM1}}{\Gamma \vdash \texttt{Expression [(+,Term (CallFactor (SymbolReference (NameReference n))) [])]} :: \texttt{Just INT}}\,\text{EXPR1} \quad (\texttt{Just INT}) = min(M(\Gamma, \texttt{faculty.multiply}(\texttt{Just INT})))}{\Gamma \vdash \texttt{Call (FieldReference faculty multiply) [Expression [(+,Term (CallFactor (SymbolReference (NameReference n))) [])]]} :: \texttt{Nothing}}\,\text{CALL4}$$

Figure 7.2: Type derivation for `faculty.multiply(n)`, `fac2.olang`, line 26

$$\dfrac{() = min(M(\Gamma, \texttt{faculty.print}()))}{\Gamma \vdash \texttt{Call (FieldReference faculty print) []} :: \texttt{Nothing}}\,\text{CALL4}$$

Figure 7.3: Type derivation for `faculty.print()`, `fac2.olang`, line 30

7.4 Limitations of the code generator implementation

The code generator implementation leaves room for improvement in two main areas.

For one, due to its sequential nature (it essentially walks the syntax tree in depth-first order), mutual recursion can only be handled in very specific cases. It is possible to translate a procedure or method that invokes one of its subprocedures, where the subprocedure invokes the calling procedure or method as well. Any other case of mutual recursion - mutually recursive top-level procedures or methods, as well as class definitions that mutually refer to each other (see the latter part of section 9) - can not be translated. Alleviating this problem requires a different approach to code generation. As a first step, it would be useful to allow backtracking to handle forward-references without mutual recursion: If code generation of a procedure fails, maybe it requires a forward reference and can only be translated later. But this is not enough for mutual recursion, since in that case there are always forward references, independent of the order of translation. To handle this, at least one of the forward references must be ignored at first, leaving placeholders for some calling instructions. Then, a second stage of translation can fill in these placeholders and complete the translation.

Additionally, the provided implementation of the code generator lacks any form of code optimization. Some optimizations like end-recursion optimization are simple enough to implement to include them in a sensible implementation of O.

CHAPTER 8

Conclusion

This thesis describes the design of a mini-language that comprises the most important features of object-oriented programming. On top of that, it delivers a simple implementation of the language, which is also described in detail, thus fulfilling the aim of the thesis introduced in chapter 1. Despite this, there is potential for improvement in several ways.

Firstly, one could think of additions to the language itself. Support for more primitive data types like boolean, fractional or character-based types could be added. Also, the grammar could be extended to allow for more flexibility regarding expressions, such as performing method calls and field references on arbitrary expressions. Furthermore, the addition of simple extensions like interfaces, abstract classes and static variables, which are common to most popular object-oriented languages, is a possibility that would not complicate O too much to still be called a mini-language.

Secondly, the implementation of the language also has some limitations. section 7.4 covered the difficulty with mutually recursive class definitions, methods and procedures, and how to extend the code generator to handle mutual recursion. Also, the possibility to incorporate some simple code optimization strategies was mentioned. The implementation of garbage collection is another point that was purposefully left out to simplify the implementation (see chapter 6).

These additions are also a step towards an idiomatic implementation of common design patterns. The fact that O shows potential in this regard was demonstrated by the implementation of the composite pattern in example 3.10. The visitor pattern as well as the alternative implementation of the example given in section 9 build on a variation of the composite pattern that relies heavily on mutual recursion and can therefore not currently be compiled using the provided implementation.

Apart from simple optimizations, the language could also be used for experimentation with other implementation concepts. Since the code generator is probably the most complicated component of the implementation, it is naturally more prone to programming errors. The provided implementation tries to lessen this problem by employing numerous automated tests that also cover the code generator. Still, it might prove beneficial to compare the existing implementation to an alternative approach using interpretation, or even try to verify its correctness by formalizing the semantics established informally for O.

With this, the author wishes to thank any reader that has read thus far and conclude this thesis.

CHAPTER 9

Appendix

FRACTRAN

In memory of John H. Conway, who passed away in 2020, this section presents an implementation of the Turing-complete FRACTRAN programming language (see [4]) in O.

A FRACTRAN program is a finite sequence of positive rational numbers $(f)_{i=1}^{k}$. Given an input N, the produced sequence of numbers N_n is given by $N_0 = N$, $N_{n+1} = f_i N_n$ where i is the minimum i such that $f_i N_n \in \mathbb{N}$. The program stops if no such i exists.

Using example 3.8 for rational numbers, along with a minimal implementation of linked lists, example 9.1 implements FRACTRAN in O by iterating through the list repeatedly. It has the programs PRIMEGAME, Fibonacci and POLYGAME pre-programmed for selection. They are constructed in the corresponding procedures to make the code more readable. There is also the option of entering a custom FRACTRAN program through the standard input.

The FRACTRAN implementation settles the question of O's computational power hinted at in example 3.4. Since using example 9.1, any FRACTRAN program can be executed, and FRACTRAN itself is Turing-complete, O is also Turing-complete.

```
USING [
    CLASS Rational(INT numerator, INT denominator)
    FIELDS INT numerator
           INT denominator
    INIT {
        IF denominator = 0 THEN {
            PRINTLNS "denominator cannot be zero!"
            ERROR
        }
        this.numerator := numerator
        this.denominator := denominator
    }
    [
        METHOD getNumerator() RETURNS INT num {
            num := this.numerator
        }

        METHOD getDenominator() RETURNS INT den {
            den := this.denominator
        }

```

```
22          METHOD multiply(INT factor) RETURNS OBJ Rational product {
23              product := Rational(factor * this.numerator, this.denominator)
24          }
25
26          METHOD isPositive() RETURNS INT isPositive {
27              isPositive := 1
28              IF this.numerator / this.denominator < 0 THEN isPositive := 0
29          }
30
31          METHOD isNatural() RETURNS INT isNatural {
32              isNatural := 0
33              IF this.isPositive() = 1 THEN {
34                  IF (this.numerator / this.denominator) * this.denominator =
    ↪  this.numerator THEN isNatural := 1
35              }
36          }
37
38          METHOD print() {
39              PRINTI this.numerator
40              PRINTS " / "
41              PRINTI this.denominator
42          }
43      ]
44
45      CLASS RationalList()
46      FIELDS INT hasHead
47             OBJ Rational head
48             INT hasNext
49             OBJ RationalList next
50      INIT {
51          this.hasHead := 0
52          this.hasNext := 0
53      }
54      [
55          METHOD print() {
56              IF this.hasHead = 1 THEN {
57                  OBJ Rational el
58                  el := this.head
59                  CALL el.print()
60                  PRINTS ", "
61                  IF this.hasNext = 1 THEN {
62                      OBJ RationalList next
63                      next := this.next
64                      CALL next.print()
65                  }
66              }
67          }
68
69          METHOD length() RETURNS INT len {
70              len := 0
71              IF this.hasHead = 1 THEN {
72                  len := 1
73                  IF this.hasNext = 1 THEN {
74                      OBJ RationalList next
75                      next := this.next
76                      len := len + next.length()
77                  }
78              }
79          }
80
81          METHOD insert(OBJ Rational element) {
82              IF this.hasHead = 1 THEN {
83                  IF this.hasNext = 1 THEN {
84                      OBJ RationalList next
85                      next := this.next
```

```
                    CALL next.insert(element)
                }
                IF this.hasNext = 0 THEN {
                    OBJ RationalList newNext
                    newNext := RationalList()
                    CALL newNext.insert(element)
                    this.next := newNext
                    this.hasNext := 1
                }
            }
            IF this.hasHead = 0 THEN {
                this.head := element
                this.hasHead := 1
            }
        }

        METHOD get(INT i) RETURNS OBJ Rational res {
            IF i < 0 THEN {
                PRINTLNS "Index out of range!"
                ERROR
            }
            IF i = 0 THEN {
                IF this.hasHead = 0 THEN {
                    PRINTLNS "Index out of range!"
                    ERROR
                }
                IF this.hasHead = 1 THEN {
                    res := this.head
                }
            }
            IF i > 0 THEN {
                IF this.hasNext = 0 THEN {
                    PRINTLNS "Index out of range!"
                    ERROR
                }
                IF this.hasNext = 1 THEN {
                    OBJ RationalList next
                    next := this.next
                    res := next.get(i - 1)
                }
            }
        }
    ]

    PROCEDURE getPrimegameProgram() RETURNS OBJ RationalList prog {
        PRINTLNS "If started with input 2, PRIMEGAME computes all prime powers of 2
↪   (among some other numbers which are not powers of 2)."
        prog := RationalList()
        CALL prog.insert(Rational(17, 91))
        CALL prog.insert(Rational(78, 85))
        CALL prog.insert(Rational(19, 51))
        CALL prog.insert(Rational(23, 38))
        CALL prog.insert(Rational(29, 33))
        CALL prog.insert(Rational(77, 29))
        CALL prog.insert(Rational(95, 23))
        CALL prog.insert(Rational(77, 19))
        CALL prog.insert(Rational(1, 17))
        CALL prog.insert(Rational(11, 13))
        CALL prog.insert(Rational(13, 11))
        CALL prog.insert(Rational(15, 2))
        CALL prog.insert(Rational(1, 7))
        CALL prog.insert(Rational(55, 1))
    }

    PROCEDURE getFibonacciProgram() RETURNS OBJ RationalList prog {
```

```
        PRINTLNS "The Fibonacci-Program computes the Fibonacci sequence f."
        PRINTLNS "Given  2 * 5 ^ (n - 1), it computes 2 ^ f(n)."
        prog := RationalList()
        CALL prog.insert(Rational(91, 33))
        CALL prog.insert(Rational(11, 13))
        CALL prog.insert(Rational(1, 11))
        CALL prog.insert(Rational(399, 34))
        CALL prog.insert(Rational(17, 19))
        CALL prog.insert(Rational(1, 17))
        CALL prog.insert(Rational(2, 7))
        CALL prog.insert(Rational(187, 5))
        CALL prog.insert(Rational(1, 3))
    }

    PROCEDURE getPolygameProgram() RETURNS OBJ RationalList prog {
        PRINTLNS "POLYGAME is a universal program - it 'enumerates' all computable
↪ functions using their respective 'catalogue numbers'."
        PRINTLNS "If c is the catalogue number of computable function f, and f(n) = m:"
        PRINTLNS "Given the number c * 2 ^ (2 ^ n), POLYGAME computes 2 ^ (2 ^ m)."
        prog := RationalList()
        CALL prog.insert(Rational(583, 559))
        CALL prog.insert(Rational(629, 551))
        CALL prog.insert(Rational(437, 527))
        CALL prog.insert(Rational(82, 517))
        CALL prog.insert(Rational(615, 329))
        CALL prog.insert(Rational(371, 129))
        CALL prog.insert(Rational(1, 115))
        CALL prog.insert(Rational(53, 86))
        CALL prog.insert(Rational(43, 53))
        CALL prog.insert(Rational(23, 47))
        CALL prog.insert(Rational(341, 46))
        CALL prog.insert(Rational(41, 43))
        CALL prog.insert(Rational(47, 41))
        CALL prog.insert(Rational(29, 37))
        CALL prog.insert(Rational(37, 31))
        CALL prog.insert(Rational(299, 29))
        CALL prog.insert(Rational(47, 23))
        CALL prog.insert(Rational(161, 15))
        CALL prog.insert(Rational(527, 19))
        CALL prog.insert(Rational(159, 7))
        CALL prog.insert(Rational(1, 17))
        CALL prog.insert(Rational(1, 13))
        CALL prog.insert(Rational(1, 3))
    }

    PROCEDURE getCustomProgram() RETURNS OBJ RationalList prog {
        PRINTLNS "You will be asked to enter each rational number, numerator and
↪ denominator separately."
        PRINTLNS "All rational numbers must be positive."
        PRINTLNS "To complete the input, enter a zero denominator."
        INT programComplete
        programComplete := 0
        INT counter
        counter := 0

        WHILE programComplete = 0 DO {
            INT den
            INT num

            PRINTS "Rational number "
            PRINTI counter
            PRINTLNS ": "
            PRINTLNS "Please enter the numerator: "
            READ num
            PRINTLNS "Please enter the denominator: "
```

```
            READ den

            IF den = 0 THEN {
                programComplete := 1
            }
            IF NOT den = 0 THEN {
                OBJ Rational newrat
                newrat := Rational(num, den)
                IF newrat.isPositive() = 0 THEN {
                    PRINTLNS "All rationals must be positive!"
                    ERROR
                }
                IF counter = 0 THEN {
                    prog := RationalList()
                    CALL prog.insert(newrat)
                }
                IF NOT counter = 0 THEN {
                    CALL prog.insert(newrat)
                }
                counter := counter + 1
            }
        }

        IF counter < 1 THEN {
            PRINTLNS "You entered an empty program!"
            ERROR
        }
    }
] DO {
    OBJ RationalList program
    INT programChoice

    PRINTLNS "Welcome to the FRACTRAN interpreter."
    PRINTLNS "Which program do you want to execute?"
    PRINTLNS "0: PRIMEGAME"
    PRINTLNS "1: Fibonacci"
    PRINTLNS "2: POLYGAME"
    PRINTLNS "3: Enter custom program interactively"
    READ programChoice
    IF programChoice < 0 THEN {
        PRINTLNS "Invalid input!"
        ERROR
    }
    IF programChoice = 0 THEN program := getPrimegameProgram()
    IF programChoice = 1 THEN program := getFibonacciProgram()
    IF programChoice = 2 THEN program := getPolygameProgram()
    IF programChoice = 3 THEN program := getCustomProgram()
    IF programChoice > 3 THEN {
        PRINTLNS "Invalid input!"
        ERROR
    }

    INT programLength
    programLength := program.length()

    INT input
    PRINTS "Input number: "
    READ input

    PRINTS "Program: "
    CALL program.print()
    PRINTLNS ""
    PRINTS "Input: "
    PRINTI input
    PRINTLNS ""
```

```
    PRINTLNS "Program output: "

    INT currentInput
    currentInput := input
    INT currentIndex
    currentIndex := 0

    WHILE currentIndex < programLength DO {
        OBJ Rational currentFrac
        currentFrac := program.get(currentIndex)
        currentFrac := currentFrac.multiply(currentInput)
        INT isNatural
        isNatural := currentFrac.isNatural()
        IF isNatural = 1 THEN {
            currentInput := currentFrac.getNumerator() / currentFrac.getDenominator()
            currentIndex := 0
            PRINTI currentInput
            PRINTLNS ""
        }
        IF isNatural = 0 THEN {
            currentIndex := currentIndex + 1
        }
    }
}
```

Listing 9.1: Example program `fractran.olang`

The list implementation could be improved by using the composite pattern introduced in example 3.10. This allows for simplifying the involved logic due to the distinction between non-empty and empty lists. In O, this can be represented by defining 3 classes `RationalListInterface` (the common interface), `RationalList` (the non-empty lists) and `RationalLeaf` (the empty lists):

```
CLASS RationalListInterface()
INIT {
    PRINTLNS "This class represents an interface and must not be instantiated!"
    ERROR
}
[
    METHOD print() {
        PRINTLNS "I am an abstract method"
        ERROR
    }

    METHOD length() RETURNS INT length {
        PRINTLNS "I am an abstract method"
        ERROR
    }

    METHOD insert(OBJ Rational element) RETURNS OBJ RationalListInterface res {
        PRINTLNS "I am an abstract method"
        ERROR
    }

    METHOD get(INT index) RETURNS OBJ Rational res {
        PRINTLNS "I am an abstract method"
        ERROR
    }
]

CLASS RationalList(OBJ Rational head)
SUBCLASSOF RationalListInterface
FIELDS OBJ Rational element
```

```
        OBJ RationalListInterface next
INIT {
    this.element := head
    this.next := RationalLeaf()
}
[
    METHOD print() {
        OBJ Rational el
        el := this.element
        CALL el.print()
        PRINTS ", "
        OBJ RationalListInterface next
        next := this.next
        CALL next.print()
    }

    METHOD length() RETURNS INT len {
        OBJ RationalListInterface next
        next := this.next
        len := 1 + next.length()
    }

    METHOD insert(OBJ Rational element) RETURNS OBJ RationalList newHead {
        OBJ RationalListInterface next
        next := this.next
        this.next := next.insert(element)

        newHead := this
    }

    METHOD get(INT i) RETURNS OBJ Rational res {
        IF i < 0 THEN {
            PRINTLNS "Index out of range!"
            ERROR
        }
        IF i = 0 THEN {
            res := this.element
        }
        IF i > 0 THEN {
            OBJ RationalListInterface next
            next := this.next
            res := next.get(i - 1)
        }
    }
]

CLASS RationalLeaf()
SUBCLASSOF RationalListInterface
INIT { PRINTS "" }
[
    METHOD print() {
        PRINTS ""
    }

    METHOD length() RETURNS INT len {
        len := 0
    }

    METHOD insert(OBJ Rational newel) RETURNS OBJ RationalList newHead {
        newHead := RationalList(newel)
    }

    METHOD get(INT i) RETURNS OBJ Rational res {
        PRINTLNS "Index out of range!"
        ERROR
```

```
        }
]
```

Apart from the minor issue that interfaces can not be represented natively in O, there is a bigger problem. Independent of the arrangement of the three classes, there is always at least one forward reference to a class defined later. Due to a shortcoming of the code generator (see chapter 7), only backward references are allowed, so the provided implementation can not be used for compiling the classes.

Bibliography

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 2005. ISBN: 9780521619462.

[3] François Bry and Norbert Eisinger. *Übersetzerbau – Abstrakte Maschinen*. Vorlesungsunterlagen/lecture notes, Institute for Informatics, Ludwig Maximilian University of Munich, Germany. 2004. URL: https://www.pms.ifi.lmu.de/publikationen/#LN_uebersetzerbau-2004.

[4] J. H. Conway. "FRACTRAN: A Simple Universal Programming Language for Arithmetic". In: *Open Problems in Communication and Computation*. Ed. by Thomas M. Cover and B. Gopinath. New York, NY: Springer New York, 1987, pp. 4–26. ISBN: 978-1-4612-4808-8. DOI: 10.1007/978-1-4612-4808-8_2.

[5] A.J.T. Davie. *Introduction to Functional Programming Systems Using Haskell*. Cambridge Computer Science Texts. Cambridge University Press, 1992. ISBN: 9780521277242.

[6] Erich Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Mar. 1995. ISBN: 978-0201633610.

[7] ISO Central Secretary. *Information technology - Vocabulary*. Standard. Geneva, Switzerland: International Organization for Standardization, May 2015. URL: https://www.iso.org/standard/63598.html.

[8] Daan Leijen. *Parsec, a fast combinator parser*. Department of Computer Science, University of Utrecht, The Netherlands. 2001. URL: https://web.archive.org/web/20120401040711/http://legacy.cs.uu.nl/daan/download/parsec/parsec.pdf.

[9] Barbara H. Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383.

[10] Daniel Maier. *Conception and Implementation of an Object-Oriented Mini-Language - provided implementation*. Version 1.0.0.0. URL: https://github.com/megmug/olang-bachelor-thesis.

[11] University of Calgary Programming Languages Laboratory Department of Computer Science. *Context Free Grammar Checker*. URL: http://smlweb.cpsc.ucalgary.ca (visited on 07/14/2022).

[12] Raphael M. Robinson. "Recursion and double recursion". In: *Bulletin of the American Mathematical Society* 54 (1948), pp. 987–993. DOI: 10.1090/S0002-9904-1948-09121-2.