The KAL Programming Language

Group S405A

May 2011

Department of Computer Science
Software
Aalborg University

AALBORG UNIVERSITY
STUDENTS STUDYREPORT

Department of Computer Science
Software
Selma Lagerlöfs Vej 300
Telephone 99 40 99 40
Fax 99 40 97 98
http://cs.aau.dk/en

Title:
The KAL Programming Language

Theme:
Language Technology

Project Period:
SW4, spring semester 2011

Project Group:
S405A

Participants:

Emil Custic

Mathias Munk Hansen

Jens Mohr Mortensen

Dan Stenholt Møller

Jesper Dalgas Zachariassen

Advisor:
Ricardo Gomes Lage

Page Count: 81

Appendix Count: 2 (12 pages)

Finished: 27/05–2011

Synopsis:

This report documents the design of the programming language KAL and the design and implementation of the KAL compiler and the KAL Virtual Machine (KVM).

The report is divided into two parts - a language specification and an implementation part. The language specification describes the evaluation criteria of KAL along with the formal syntax and semantics of KAL. The implementation part describes the design and development process of the KAL compiler and the KVM.

PREFACE

This report was written as a 4th semester project by a group of students from the Department of Computer Science at Aalborg University (AAU). The report will document the process of designing a programming language and implementing a compiler and a virtual machine for it.

When references are used in the report they will be referred to in the format [abc] with a corresponding entry in the bibliography (located in the back of the report). For larger works a range of relevant pages will also be specified in the format [p. 10-11]. Figures and tables will be referred to in the format [2.1], where the first number refers to the chapter in which the figure/table is placed and the second number is the actual number of the figure/table.

The reader is expected to have basic knowledge of programming language concepts, operational semantics and Java.

The formal semantics of KAL are based on the textbook *Transitions and Trees* [Hü10] from the course *Syntax and Semantics*. The implementation of the compiler and virtual machine is based on the textbook *Programming Language Processors in Java* [WB00] from the course *Language and Compiler Construction*. Both courses were held during this semester.

The CD-ROM[1] included with this report contains the complete source code of the compiler and virtual machine as well as this report.

Finally, we would like to thank our advisor Ricardo Gomes Lage for the constructive feedback.

[1] Available online at: `http://pushittothelimit.dk/sw4/cdrom.zip`

CONTENTS

CHAPTER 1

INTRODUCTION

The purpose of this report is to document the design of the programming language *KAL* as well as the implementation of the KAL compiler and the *KAL Virtual Machine* (KVM) to run the compiled code on. Our goal is to create a simple, structured general-purpose programming language, and to learn about relevant theory and implementation techniques for language processors. Furthermore, it is our goal to meet the formal requirements for this project as defined in the study regulation for SW4 [Und09]. These state that at the end of the SW4 semester project, the student must be able to:

- Document knowledge and overview of the applied techniques and terms in language design and translator construction.
- Describe, analyze and implement a translator or interpreter for a concrete programming language or an extension for an existing programming language.
- Account for the individual phases in a translator and the connection between them.
- Account for the applied implementation techniques in the constructed translator/interpreter.
- Use correct terminology.
- Reason as a software engineer using the applied terms and techniques.

1.1 Report Structure

The structure of this report is as follows:

- In Chapter 2 on page 3 we specify the criteria and formal syntax and semantics for KAL.
- In Chapter 3 on page 19 we specify the criteria for the KAL-compiler and KVM and describe the implementation of the different components in detail.

- In Chapter 4 on page 65 we reflect on the criteria previously defined and how the choices made in the design of KAL and the implementation affected them.

CHAPTER 2

LANGUAGE SPECIFICATION

This chapter will serve as the specification of KAL, containing all the formal requirements. We begin by establishing a set of basic evaluation criteria. Based on these criteria we specify the syntax, contextual constraints and semantics of KAL. We finish the chapter by providing an example of a small program written in KAL.

2.1 Evaluation Criteria

To aid us in making design choices we have defined three evaluation criteria for KAL and assigned a priority of either low, medium or high to each of them. We chose to prioritize readability as high because KAL is targeted at intermediate level programmers. The evaluation criteria and their priorities are shown in Table 2.1. Evaluation criteria that are implementation-specific, such as performance, are discussed later in Section 3.1 on page 20.

Criteria	**Definition**	**Priority**
Readability	The ease with which a program can be read and understood.	High
Writability	The ease with which a program can be written for the target problem domain of the programming language.	Medium
Reliability	A programming language is reliable if it performs as intended under all conditions.	Medium

Table 2.1: Evaluation criteria and priorities for KAL.

Table 2.2 [Seb09, p. 26] shows these three evaluation criteria along with their associated characteristics. The table also illustrates the fact that writability is related to readability [Seb09, p. 32], because a programmer writing a program will often have to look back at the code already written. Likewise, reliability is related to both readability and writability, because a programming language with poor writability will often lead to unnatural solutions, which are more error-prone [Seb09, p. 35].

Characteristic	**Readability**	**Writability**	**Reliability**
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Table 2.2: Evaluation criteria and associated characteristics.

Simplicity

To increase readability, KAL will only have a small set of basic components such as control structures. In a language with a large number of basic components, a programmer will often only learn a subset of the language. Readability is decreased when another programmer has

learned a different subset of the language. Knowing only a portion of a language might also cause a programmer to use a feature in a situation where another feature would be preferable, which decreases writability [Seb09, p. 27].

To increase readability further, we will avoid *feature multiplicity* to the extent possible. Feature multiplicity in a programming language means that a certain task can be accomplished in several ways. As an example of feature multiplicity, many programming languages allow the use of `x += y` instead of `x = x + y`.

Orthogonality

An orthogonal programming language is one that has a small number of primitive constructs and a homogeneous set of legal ways to combine them. The constructs of KAL should be allowed to be combined and used freely, which will enhance both readability and writability. For example, a function call should be legal to use in an arithmetic expression, such as `x = func(y) + 5`.

Data types

KAL will have four primitive data types: Integers (`int`), floating point numbers (`float`), characters (`char`) and truth values (`bool`). The only composite data type in KAL is arrays. Data types increase the readability of the language. For example, `isDone = true` is intuitively more clear than `isDone = 1` [Seb09, p. 30].

Syntax design

In KAL there are no restrictions on the length of identifiers. The goal is to allow programmers to choose descriptive names for variables, functions and procedures, which increases readability. Identifiers are case-sensitive in order to avoid confusion such as referencing the same variable using different case.

For grouping statements we chose a matching pair of `{...}` instead of `begin...end` or `begin if...end if`. One might argue that the latter is more readable because it states explicitly which statement is being ended, but we argue that the trade-off in writability caused by the amount of extra characters to be typed exceeds the gain in readability. Another reason for our choice is that many people are familiar with this syntax since it is used in many popular programming languages, such as C#, Java and C. With proper indentation the difference in readability is negligible.

Support for abstraction

KAL supports two kinds of abstraction, namely functions and procedures. This increases writability by allowing the programmer to reuse common algorithms several places in the code without copying the entire implementation of the algorithm.

Expressivity

We have chosen not to implement various shorthand notations for commonly used features. The reason for this is that it would create feature multiplicity, which reduces readability.

Type checking

KAL will be *statically typed*, meaning that all type errors will be detected at compile-time, as opposed to *dynamically typed* where type errors cannot be detected until run-time. This increases reliability by ensuring that the program will not produce a run-time error or an unexpected result due to an invalid operation.

Exception handling

KAL will support various kinds of run-time exceptions, such as `IndexOutOfBounds` for arrays and `ZeroDivide` for divisions by zero. Run-time exceptions increase the reliability by halting the execution of the program instead of producing an unexpected result.

Restricted aliasing

Aliasing means that a location in memory can be accessed using different identifiers. KAL will restrict aliasing, which increases reliability. For example, in the code `x = 5; y = x; x = 10;` the final value of `y` will be 5 in a language that does restrict aliasing, and 10 in one that does not.

2.2 Syntax

In this section the grammar of KAL will be formally specified. This section has been divided into three parts. The first part is the grammar expressed in *Backus-Naur Form* (BNF). The second part is the grammar expressed in *Extended Backus-Naur Form* (EBNF), which is used later in the implementation of the KAL compiler. The third part is the *lexicon* of KAL used by the *syntactic analyzer* (see Section 3.3 on page 27) also expressed in EBNF.

2.2.1 BNF

A grammar in BNF consists of a number of *production rules* on the form $N \rightarrow \alpha$, where the left-sides consist of *nonterminal symbols* and the right-sides consist of nonterminal symbols and *terminal symbols* (emboldened). The symbol $\rightarrow$ may be read as "consists of", | may be read as "or" and ε means an empty string [WB00, p. 7]. The following is the grammar of KAL expressed in BNF:

(1)	Program	→	Import Declaration
(2)	Import	→	Import SingleImport
		|	SingleImport
		|	ε

(3) SingleImport → **import** Identifier **;**

(4) Declaration → Declaration VariableDeclaration **;**
| Declaration ConstantDeclaration **;**
| Declaration ArrayDeclaration **;**
| Declaration ProcedureDeclaration
| Declaration FunctionDeclaration
| VariableDeclaration **;**
| ConstantDeclaration **;**
| ArrayDeclaration **;**
| ProcedureDeclaration
| FunctionDeclaration

(5) VariableDeclaration → TypeDenoter Identifier

(6) ConstantDeclaration → **constant** TypeDenoter Identifier = Literal

(7) ArrayDeclaration → **array** TypeDenoter Identifier [IntegerLiteral]
| **array char** Identifier [ ] = String

(8) FunctionDeclaration → **function** TypeDenoter Identifier **(** FormalParameterSequence **)** { Command }
| **function** TypeDenoter Identifier **(** FormalParameterSequence **) ;**

(9) ProcedureDeclaration → **procedure** Identifier **(** FormalParameterSequence **)** { Command }
| **procedure** Identifier **(** FormalParameterSequence **) ;**

(10) FormalParameterSequence → FormalParameterSequence **,** SingleFormalParameter
| SingleFormalParameter
| ε

(11) SingleFormalParameter → VariableDeclaration
| ArrayFormalDeclaration

(12) ArrayFormalDeclaration → **array** TypeDenoter Identifier [ ]

(13) Command → Command SingleCommand
| SingleCommand

(14) SingleCommand → VariableDeclaration **;**
| ArrayDeclaration **;**
| Identifier = Expression **;**
| Identifier [Expression] = Expression **;**
| Identifier **(** ActualParameterSequence **) ;**
| **if (** Expression **)** { Command }
| **if (** Expression **)** { Command } **else** { Command }
| **while (** Expression **)** { Command }
| **return ;**
| **return** Expression **;**

(15) Expression → Expression Operator SingleExpression
| SingleExpression

(16) SingleExpression → Identifier

	|	Identifier **(** ActualParameterSequence **)**
	|	Identifier **[ ]**
	|	Identifier **[** Expression **]**
	|	**(** Expression **)**
	|	Operator SingleExpression
	|	Literal
(17) ActualParameterSequence	→	ActualParameterSequence **,** SingleActualParameter
	|	SingleActualParameter
	|	ε
(18) SingleActualParameter	→	Expression

2.2.2 EBNF

After expressing the grammar of KAL in BNF, we transformed it into EBNF, which is BNF with the addition of *regular expressions* such as * (meaning zero or more times), parentheses for grouping etc. The reason for this transformation is that a grammar expressed in EBNF is more suitable for *recursive-descent parsing* (see Section 3.3.2 on page 29) than one expressed in BNF, because every production rule can be implemented as a method in the parser [WB00, p. 93–94].

In order for recursive-descent parsing to be possible the grammar must be LL(1), meaning that the parser only needs to look one token ahead to decide which action to take [WB00, p. 104]. In some cases we had to add *noise words* in order to achieve this. For example, the words "constant", "array" and "function" have been added to differentiate between the rules VariableDeclaration, ConstantDeclaration, ArrayDeclaration and FunctionDeclaration, since they all begin with TypeDenoter.

When transforming the grammar from BNF to EBNF we used three rules: *Left factorization*, *elimination of left recursion* and *substitution of nonterminal symbols*. For example, using left factorization we can rewrite part of rule (14) of the grammar in BNF to:

SingleCommand	→	**if (** Expression **) {** Command **} (** (**else {** Command **})** | ε **)**

By eliminating left recursion on rule (13) of the grammar in BNF we get:

Command	→	SingleCommand SingleCommand*

By eliminating left recursion and substituting nonterminal symbols on rule (17) of the grammar in BNF we get:

ActualParameterSequence	→	Expression (**,** Expression)*
	|	ε

The following is the grammar of KAL expressed in EBNF, which was made by applying left factorization, eliminating left recursion and substituting nonterminal symbols in the grammar in BNF wherever possible:

---

(1)	Program	→	Import* Declaration Declaration*
(2)	Import	→	**import** Identifier **;**
(3)	Declaration	→	VariableDeclaration **;**
		|	ConstantDeclaration **;**
		|	ArrayDeclaration **;**
		|	ProcedureDeclaration
		|	FunctionDeclaration
(4)	VariableDeclaration	→	TypeDenoter Identifier
(5)	ConstantDeclaration	→	**constant** TypeDenoter Identifier = Literal
(6)	ArrayDeclaration	→	**array** TypeDenoter Identifier [IntegerLiteral]
		|	**array char** Identifier [ ] = String
(7)	FunctionDeclaration	→	**function** TypeDenoter Identifier **(** FormalParameterSequence **)** (({ Command }) | **;**)
(8)	ProcedureDeclaration	→	**procedure** Identifier **(** FormalParameterSequence **)** (({ Command }) | **;**)
(9)	FormalParameterSequence	→	SingleFormalParameter (**,** SingleFormalParameter)*
		|	ε
(10)	SingleFormalParameter	→	VariableDeclaration
		|	ArrayFormalDeclaration
(11)	ArrayFormalDeclaration	→	**array** TypeDenoter Identifier **[ ]**
(12)	Command	→	SingleCommand (SingleCommand)*
(13)	SingleCommand	→	VariableDeclaration **;**
		|	ArrayDeclaration **;**
		|	Identifier ((([Expression] | ε) = Expression) | (**(** ActualParameterSequence **)**)) **;**
		|	**if (** Expression **)** { Command } ((**else** { Command }) | ε)
		|	**while (** Expression **)** { Command }
		|	**return** (Expression | ε) **;**
(14)	Expression	→	SingleExpression (Operator SingleExpression)*
(15)	SingleExpression	→	Identifier ((**(** ActualParameterSequence **)**) | ([Expression]) | (**[ ]**) | ε)
		|	**(** Expression **)**
		|	Operator SingleExpression

	|	Literal
(16) ActualParameterSequence	→	Expression (**,** Expression)*
	|	ε

2.2.3 Lexicon

The lexicon is used by the *syntactic analyzer*. In the lexicon "Graphic" means any character and "EOL" is an end-of-line symbol. The following is the lexicon of KAL expressed in EBNF:

(1) Program	→	(Token | Comment | Separator)*
(2) Token	→	Identifier
	|	Operator
	|	Literal
	|	TypeDenoter
	|	**if** | **else** | **while** | **function** | **procedure** | **return** | **import**
	|	**(** | **)** | **{** | **}** | **[** | **]** | **;** | **,** | **'** | **"** | **=**
(3) Comment	→	# Graphic* EOL
(4) Separator	→	Space | Tab | EOL
(5) Identifier	→	Letter (Letter | Digit)* (**.** Letter | ε) (Letter | Digit)*
(6) Operator	→	**+** | **-** | ***** | **/** | **%** | **==** | **!=** | **!** | **<** | **<=** | **>** | **>=** | **&&** | **||**
(7) Literal	→	IntegerLiteral | CharacterLiteral | FloatLiteral | BooleanLiteral
(8) IntegerLiteral	→	Digit Digit*
(9) CharacterLiteral	→	**'** ASCII **'**
(10) FloatLiteral	→	Digit Digit* **.** Digit Digit*
(11) BooleanLiteral	→	**true** | **false**
(12) TypeDenoter	→	**int** | **char** | **bool** | **float**
(13) String	→	**"** ASCII* **"**
(14) Letter	→	**a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**
(15) Digit	→	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

2.3 Semantics

We have described the formal semantics of KAL using *structural operational semantics*. We chose a formal semantics over an informal semantics because it allow us to express the semantics unambiguously.

The semantics of KAL are described as a *big-step semantics*, meaning that a single transition describes an entire computation. We chose a big-step semantics because it is easier to create than a *small-step semantics* and because there is no parallelism in KAL, so there is no need for a small-step semantics [Hü10, p. 31].

2.3.1 Informal Scope Rules

In KAL there are two different kinds of block structures: One for control structures and one for routines.

The block structure applied in KAL for routines is called a *flat block structure* [WB00, p. 139–142]. This means that there are two block levels, a global and a local level. This is shown in the grammar as routines can only be declared globally, which yields a flat block structure.

The block structure applied for control structures is called a *nested block structure* [WB00, p. 142–145]. This means that control structures can be nested within control structures, as shown in the grammar.

In KAL the open curly bracket ({) is used to indicate that a new block has started, and the close curly bracket (}) is used to indicate when a block has ended. The outermost block in KAL is at scope level zero. When a routine is declared the scope level is increased to one. If a control structure is used the scope level will be incremented by one.

The scope rules of KAL are as follows:

- No declared identifier may be redeclared in the same block or blocks encapsulated by it. However, locally declared identifiers may be declared in blocks not encapsulated by it.
- For every applied occurrence of an identifier in a block, there must be a corresponding declaration in the block it was used in or in a block encapsulating it.

2.3.2 Transition Rules

In this section some of our transition rules will be described. All of the transition rules can be seen in Appendix A on page 69. In the transition rules of KAL we use the following names to represent different syntactic categories:

- n - Numerals.
- v - Values.
- $x \in$ **Vars** - Variable names.
- $r \in$ **Arrays** - Array names.

- $c \in$ **Consts** - Constant names.
- $p \in$ **ProcNames** - Procedure names.
- $f \in$ **FuncNames** - Function names.
- $T \in \{$`int`, `float`, `char`, `bool`$\}$ - Types.
- $a \in$ **ArExp** - Arithmetic expressions.
- $b \in$ **BoolExp** - Boolean expressions.
- $e \in$ **ArExp** $\cup$ **BoolExp**.
- $C \in$ **Com** - Commands.
- *VD* - Element of variable declarations.
- *AD* - Element of array declarations.
- *CD* - Element of constant declarations.
- *PD* - Element of procedure declarations.
- *FD* - Element of function declarations.

Environment-Store Model

To represent the storage of a computer we use the *environment-store model* [Hü10, p. 80–82]. This model represents how a variable is bound to a storage cell (also called a *location*), and that the value of the variable is the content of that storage cell. All locations are denoted by **Loc** and a single location by $l \in$ **Loc**. We assume that locations are natural numbers so that **Loc** $= \mathbb{N}$. Since all locations are natural numbers, we can define a function that for any location, l, points to the location immediately following it as new : **Loc** $\rightarrow$ **Loc**, where new $l = l + 1$.

To declare new variables and bind them to new locations we define a pointer, next, which points to the next available location.

Furthermore, we define the set of stores to be the mappings from locations to values **Sto** = **Loc** $\rightharpoonup \mathbb{R}$, where $sto \in$ **Sto**.

In the transition rules we use the following environments:

- $env_V \in$ **EnvV** - Variable declarations.
- $env_A \in$ **EnvA** - Array declarations.
- $env_C \in$ **EnvC** - Constant declarations.
- $env_P \in$ **EnvP** - Procedure declarations.
- $env_F \in$ **EnvF** - Function declarations.
- env_E - Shorthand notation for "env_V, env_A, env_C, env_F" used throughout the transition rules.

Arithmetic Expressions

The transition rule for addition in KAL can be seen in Table 2.3. The rule describes that if a_1 and a_2 evaluates to v_1 and v_2 respectively, using any of the rules from **ArExp**, then a_1 + a_2 evaluates to v where $v = v_1 + v_2$.

$$\text{[ADD]} \qquad \frac{env_E,\ sto \vdash a_1 \rightarrow_a v_1 \quad env_E,\ sto \vdash a_2 \rightarrow_a v_2}{env_E,\ sto \vdash a_1 + a_2 \rightarrow_a v}$$

where $v = v_1 + v_2$

Table 2.3: Transition rule for addition [**ArExp**].

Boolean Expressions

The transition rule for logical-and in KAL can be seen in Table 2.4. There are two rules for logical-and: One for true and one for false. The rule [AND-TRUE] states that if both b_1 and b_2 evaluate to true, using any of the rules from **BoolExp**, then the expression b_1 **&&** b_2 evaluates to true. The rule [AND-FALSE] states that if either b_1 or b_2 evaluates to false, then the expression b_1 **&&** b_2 evaluates to false.

$$\text{[AND-TRUE]} \qquad \frac{env_E,\ sto \vdash b_1 \rightarrow_b \text{true} \quad env_E,\ sto \vdash b_2 \rightarrow_b \text{true}}{env_E,\ sto \vdash b_1 \ \&\& \ b_2 \rightarrow_b \text{true}}$$

$$\text{[AND-FALSE]} \qquad \frac{env_E,\ sto \vdash b_1 \vee b_2 \rightarrow_b \text{false}}{env_E,\ sto \vdash b_1 \ \&\& \ b_2 \rightarrow_b \text{false}}$$

Table 2.4: Transition rules for logical-and [**BoolExp**].

Variable Declarations

The transition rule for variable declarations in KAL can be seen in Table 2.5. The declaration of a variable in KAL is done by binding l to the next available location and then binding x to this location. The next variable is then updated to point to the next available location again by using the function new. The variable environment env_V is updated to reflect the newly declared variable, while the store remains unchanged since no assignment is done.

$$\text{[VAR-DECL]} \qquad \frac{\left\langle VD,\ env_V\left[x \mapsto l\right]\left[\text{next} \mapsto \text{new}\ l\right],\ sto\right\rangle \rightarrow_{VD} (env'_V,\ sto)}{\left\langle T\ x,\ env_V,\ sto\right\rangle \rightarrow_{VD} (env'_V,\ sto)}$$

where $l = env_V$ next

Table 2.5: Transition rule for variable declarations.

Variable Assignments

The transition rule for variable assignments in KAL can be seen in Table 2.6. When a variable assignment is made the contents of l is updated to v, where l is the location of x found in the variable environment and v is the result of evaluating e.

$$\text{[VAR-ASS]} \qquad env_E \vdash \left\langle x = e,\ sto \right\rangle \rightarrow sto[l \mapsto v]$$

where $env_E,\ sto \vdash e \rightarrow_e v$
and $env_V\ x = l$

Table 2.6: Transition rule for variable assignments [**Com**].

If-Else Statements

The transition rules for if-else statements in KAL can be seen in Table 2.7. There are two rules for if-else statements: One if the conditional evaluates to true, and one if the conditional evaluates to false. The rule [IF-ELSE-TRUE] states that if the conditional b evaluates to true, then C_1 will be executed, which will update the store. If b evaluates to false, the same will happen, except that C_2 will be executed instead.

$$\text{[IF-ELSE-TRUE]} \qquad \frac{env_E \vdash \left\langle C_1,\ sto \right\rangle \rightarrow sto'}{env_E \vdash \left\langle \texttt{if}(b)\ \{C_1\}\ \texttt{else}\ \{C_2\},\ sto \right\rangle \rightarrow sto'}$$

if $env_E,\ sto \vdash b \rightarrow_b \text{true}$

$$\text{[IF-ELSE-FALSE]} \qquad \frac{env_E \vdash \left\langle C_2,\ sto \right\rangle \rightarrow sto'}{env_E \vdash \left\langle \texttt{if}(b)\ \{C_1\}\ \texttt{else}\ \{C_2\},\ sto \right\rangle \rightarrow sto'}$$

if $env_E,\ sto \vdash b \rightarrow_b \text{false}$

Table 2.7: Transition rules for if-else statements [**Com**].

2.3.3 Type Rules

In this section some of our type rules will be described. All of the type rules can be seen in Appendix A on page 69.

In the type rules we use the function E, which is defined as E : **Vars** $\cup$ **Consts** $\cup$ **Arrays** $\cup$ **FuncNames** $\rightharpoonup T$.

Arithmetic Expressions

The type rule for addition is shown in Table 2.8, and states that if both expressions e_1 and e_2 are of type T, then the result of the addition is also of type T. This means that if e_1 is of type `int` and e_2 is of type `int`, then the result of the addition is also of type `int`.

$$\text{[ADD]} \qquad \frac{E \vdash e_1 : T \; E \vdash e_2 : T}{E \vdash e_1 \texttt{ + } e_2 : T}$$

Table 2.8: Type rule for addition.

Boolean Expressions

The type rule for logical-and is shown in Table 2.9. The rule states that both expressions e_1 and e_2 must evaluate to values of type `bool`. If they do, e_1`&&`e_2 will evaluate to a value of type `bool`.

$$\text{[AND]} \qquad \frac{E \vdash e_1 : \texttt{bool} \; E \vdash e_2 : \texttt{bool}}{E \vdash e_1 \texttt{ \&\& } e_2 : \texttt{bool}}$$

Table 2.9: Type rule for logical-and.

Variable Declarations

The type rule for variable declarations is shown in Table 2.10. The rule states that when a variable x is declared it is mapped to a type T. This means that when x is applied its type can be checked.

$$\text{[VAR-DECL]} \qquad \frac{E[x \mapsto T]}{E \vdash T \; x : \text{ok}}$$

Table 2.10: Type rule for variable declarations.

Variable Assignments

The type rule for variable assignments is shown in Table 2.11. This rule states that in a variable assignments the expression e must have the same type as the variable x. If this is the case the assignment is well-typed.

$$\text{[VAR-ASS]} \qquad \frac{E \vdash x : T \; E \vdash e : T}{E \vdash x \texttt{ = } e : \text{ok}}$$

Table 2.11: Type rule for variable assignments.

If-Else Statements

The type rule for if-else statements is shown in Table 2.12. This rule states the conditional e must evaluate to a value of type `bool` and both commands C_1 and C_2 must be well-typed.

$$\text{[IF-ELSE]} \qquad \frac{E \vdash e : \textbf{bool}\ E \vdash C_1 : \text{ok}\ E \vdash C_2 : \text{ok}}{E \vdash \textbf{if}(e)\ \{C_1\}\ \textbf{else}\ \{C_2\} : \text{ok}}$$

Table 2.12: Type rule for if-else statements.

2.4 Code Example

Listing 2.2 shows an example of the quicksort algorithm [CLRS09] implemented in KAL. The algorithm takes an unsorted array of integers and sorts it in ascending order. The `StdIO` library for KAL can be seen in Appendix B on page 79. The output from running the program is shown in Listing 2.1.

```
Unsorted:
4 2 7 3 1 0 6 2 8 5
Sorted:
0 1 2 2 3 4 5 6 7 8
```

Listing 2.1: Quicksort algorithm output.

```
1  import StdIO;
2
3  # Quicksort algorithm implemented in KAL.
4  procedure Quicksort(array int input[], int LowerIndex, int UpperIndex)
5  {
6     int i;
7     int j;
8     int x;
9     int Tmp;
10
11    i = LowerIndex;
12    j = UpperIndex;
13    x = input[(LowerIndex + UpperIndex) / 2];
14
15    while (i <= j)
16    {
17       while (input[i] < x) { i = i + 1; }
18
19       while (input[j] > x) { j = j - 1; }
20
21       if (i <= j)
22       {
23          Tmp = input[i];
24          input[i] = input[j];
25          input[j] = Tmp;
26          i = i + 1;
27          j = j - 1;
28       }
29    }
30
31    if (LowerIndex < j) { Quicksort(input[], LowerIndex, j); }
32
33    if (i < UpperIndex) { Quicksort(input[], i, UpperIndex); }
```

```
34 }
35
36 procedure Run()
37 {
38    array int a[10];
39    a[0] = 4; a[1] = 2; a[2] = 7; a[3] = 3; a[4] = 1;
40    a[5] = 0; a[6] = 6; a[7] = 2; a[8] = 8; a[9] = 5;
41
42    array char unsorted[] = "Unsorted:";
43    StdIO.PrintString(unsorted[]);
44
45    int i;
46    i = 0;
47
48    while(i < 10)
49    {
50       WriteInt(a[i]);
51       WriteChar(' ');
52       i = i + 1;
53    }
54
55    Quicksort(a[], 0, 9);
56
57    StdIO.PrintNewLine();
58    array char sorted[] = "Sorted:";
59    StdIO.PrintString(sorted[]);
60
61    i = 0;
62
63    while(i < 10)
64    {
65       WriteInt(a[i]);
66       WriteChar(' ');
67       i = i + 1;
68    }
69 }
```

Listing 2.2: Quicksort algorithm implemented in KAL.

CHAPTER 3

IMPLEMENTATION

In this chapter we present all the relevant stages of the implementation of the KAL compiler and the KVM, including program architecture and criteria. Each part of the development process will be explained in separate sections and presented in order of dependency (components which depend on the implementation of other components will be explained after the components they depend on).

3.1 Evaluation Criteria

Table 3.1 shows some common evaluation criteria for software development and their individual priorities in the implementation of the KAL compiler and KVM.

Criterion	Very important	Important	Less important	Irrelevant
Portability	•			
Correctness	•			
Flexibility		•		
Comprehensibility		•		
Performance			•	

Table 3.1: KAL compiler evaluation criteria.

Portability

Portability is considered very important as there is an increasing number of available operating systems and development platforms, and programs written in KAL should be able to run on most. To increase the portability of programs written in KAL, each program will be interpreted by the KVM. The trade-off is that using an interpreter will impact performance as it imposes an amount of overhead which will slow down the execution of KAL programs. To compensate for this all KAL programs are compiled to *KAL Intermediate Language* (*KIL*) which is a low-level language designed for *iterative interpretation* (see Section 3.5.2 on page 47). This allows the KVM to perform faster interpretation than if it had to interpret the KAL code directly, which would require a *recursive interpreter* [WB00, p. 39]. The KVM is much smaller and easier to implement than the KAL compiler and as such will be easier to port to other platforms.

Correctness

Correctness is considered very important because the KAL compiler must generate code that performs according to the semantics defined in Section 2.3 on page 11 to be useful. By defining formal semantics for KAL we avoid semantic ambiguity allowing us to correctly implement the semantics.

Flexibility

Since this is an educational project, we want to be able to easily modify the KAL compiler in case we want to expand KAL or test different compiler implementation techniques. This is done by separating the compiler into components and programming the components to be *loosely coupled* but *highly cohesive* [MMMNS00, p. 271]. Loose coupling means that each component operates independently of other components. This allows us to replace individual

components of the compiler without affecting the others, giving us a higher degree of flexibility. High cohesiveness means that the constituent classes of components are conceptually related and that each component performs key operations. This increases the comprehensibility of the program as it makes it easier to understand the purpose of each component and its constituent classes.

Comprehensibility

We are 5 students collaborating on the implementation of the KAL compiler and for that reason it is important for us that the code is as easy to understand as possible. As explained above this is done by dividing the KAL compiler into components that are cohesive. Furthermore comprehensibility is increased by ensuring that names of objects, variables and methods in the source code are as self-explanatory as possible, and that comments are added where necessary.

Performance

There is a trade-off between compile-time performance and flexibility and run-time performance and portability in the KAL compiler, so performance has been deprioritized in favor of flexibility and portability.

3.2 Architecture

The architecture of the KAL compiler and KVM is shown in Figure 3.1 on the next page. Two machines are illustrated, both running the Java Virtual Machine (JVM). The machines are separated in the illustration although they could be a single machine in practice. The illustration reflects the classes in the implementation of the KAL compiler and KVM and will act as a foundation for the structure of most of the remainder of this chapter.

The process of compiling a source program written in KAL to running it on the KVM is as follows. First, the source program is parsed by the *Syntactic Analyzer*, which checks if it conforms with the syntactic rules defined in Section 2.2 on page 6 and creates a tree representation of the source program called an *Abstract Syntax Tree* (AST), which is described in Section 3.2.3 on page 25. If, at any point in this process, a syntactic error in the source code is discovered, the compilation process will terminate and an error log will be presented. Syntactic analysis is explained in detail in Section 3.3 on page 27.

On success, the *Contextual Analyzer* will traverse the AST, checking if the source program conforms with the contextual constraints such as scope and type rules. If the source program contains contextual errors, the compilation process will terminate and an error log will be presented. Contextual analysis is explained in detail in Section 3.4 on page 33.

Finally, the *Code Generator* will traverse the AST and translate the source program into KIL code, which can be run by the KVM. Code generation is explained in detail in Section 3.6 on page 54.

To run the KIL program it is loaded into the KVM where it will be interpreted. The KVM is

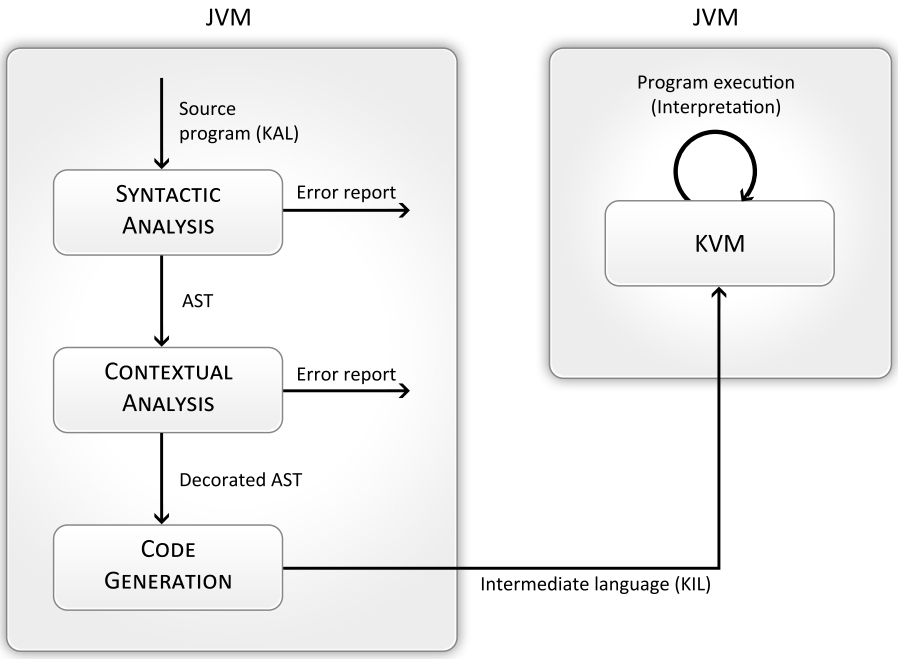

Figure 3.1: An overview of the KAL compiler and KVM.

explained in detail in Section 3.5 on page 42.

3.2.1 Language Processing Strategy

In this section we will illustrate our language processing strategy using *tombstone diagrams* [WB00, p. 26–54]. The dark tombstones represent the Java Compiler or the JVM as shown in Figure 3.2, and the light tombstones represent the parts we are implementing. This color separation has been made in order to illustrate a clear difference between what we are implementing and what is simply a tool in this process.

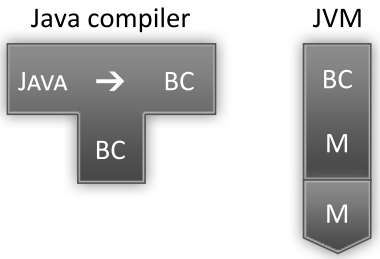

Figure 3.2: Tombstone diagrams for the Java Compiler and the JVM.

The tombstone diagrams are divided into four parts with the first illustrating the process of

compiling the KAL to KIL compiler (Figure 3.3), the second illustrating the process of compiling KAL to KIL (Figure 3.4 on the following page), the third illustrating the process of compiling the KVM (Figure 3.5 on the next page), and the fourth illustrating the process of how a program written in KIL is interpreted (Figure 3.6 on page 25).

KAL to KIL Compilation and Compiler Compilation

Figure 3.3 illustrates the process of compiling the KAL to KIL compiler. As can be seen, the KAL to KIL compiler is written in Java and is then compiled by the Java Compiler to a KAL to KIL compiler written in Java bytecode. This makes the KAL to KIL compiler written in Java bytecode platform independent, since it can be run on any platform running the JVM.

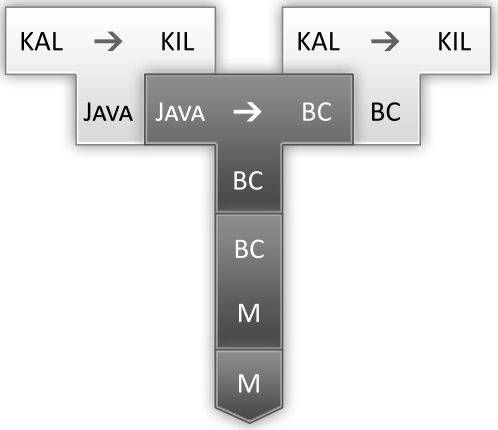

Figure 3.3: A tombstone diagram of the KAL to KIL compiler compilation process.

Figure 3.4 on the following page illustrates the process of compiling KAL to KIL. As can be seen, the KAL to KIL compiler written in Java bytecode is used to compile a program written in KAL to a semantically equivalent program written in KIL.

KIL Interpreter Compilation and Interpretation

Figure 3.5 on the next page illustrates the process of compiling the KVM. As can be seen, KVM is written in Java and is then compiled by the Java Compiler to Java bytecode. As with the KAL to KIL compiler, the KVM is also platform independent and can be run on any platform running the JVM.

Figure 3.6 on page 25 illustrates the process of running a program written in KIL on the KVM. As can be seen, the program is running on the KVM which is running on the JVM. This implies that KIL is also platform independent.

3.2.2 Compilation Passes

When designing the architecture of a compiler it is important to decide whether a compiler should be single-pass or multi-pass as both have advantages and disadvantages. The KAL to

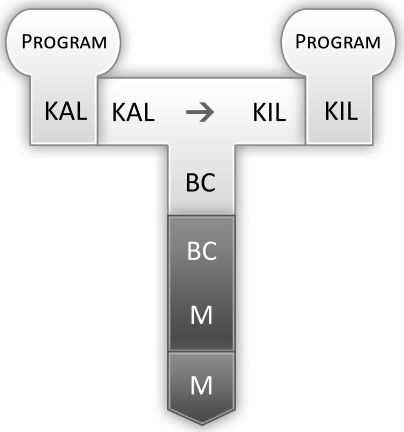

Figure 3.4: A tombstone diagram of the KAL to KIL compilation process.

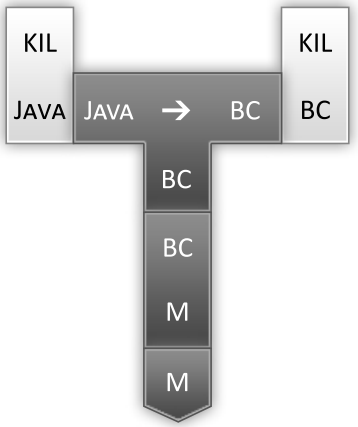

Figure 3.5: A tombstone diagram of the KVM compilation process.

KIL compiler is designed to be a multi-pass compiler in order to make it as flexible as possible. In a multi-pass compiler each of the individual modules (syntactic analysis, contextual analysis and code generation) only depend on the AST. In a single-pass compiler, the syntactic analyzer has the task to not only analyze the syntax of the source program, but also to coordinate the contextual analyzer and the code generator. A modular design enables us to easily replace the implementation of a module. A multi-pass compiler allows the AST to be traversed in any order and as many times as needed. Furthermore, because the compiler is divided into several independent modules, members of the group are able to work independent of each other.

There are other factors to take into consideration when deciding between multi or single-pass compilation, such as compile-time performance, code optimization and various restrictions

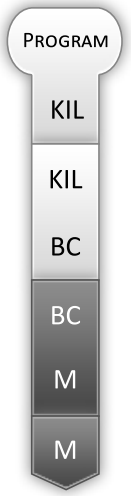

Figure 3.6: A tombstone diagram of the KIL interpretation process.

in the source language. These are not considered relevant for our decision. For example, the compile and run-time performance are both of low priority as can be seen in Section 3.1 on page 20 [WB00, p. 63–72].

3.2.3 Abstract Syntax Tree

The implementation of the AST is based on the grammar expressed in EBNF. In this project, the AST is implemented as an abstract class called `AST`, along with a number of abstract and concrete classes extending the `AST` class and implementing the rules of the grammar. For example, the concrete class `Program` has the instance variable `dASTList`, which is a list of declarations. This conforms with the first rule of the grammar.

As another example, the abstract class `Declaration` (see Listing 3.1) is an abstraction of the concrete classes `ArrayDeclaration`, `ArrayFormalDeclaration`, `ConstantDeclaration`, `VariableDeclaration`, `FunctionDeclaration`, `BinaryOperatorDeclaration`, `UnaryOperatorDeclaration` and `ProcedureDeclaration`. Listing 3.2 on the next page shows the implementation of the `FunctionDeclaration` class.

```
1 public abstract class Declaration extends AST {
2    public boolean duplicateDecl;
3    public boolean forwardDecl;
4
5    public Declaration() {
6       duplicateDecl = false;
7       forwardDecl = false;
8    }
9 }
```

Listing 3.1: The abstract `Declaration` class.

```
1  public class FunctionDeclaration extends Declaration {
2     public TypeDenoter tdAST;
3     public Identifier iAST;
4     public FormalParameterSequence fpsAST;
5     public Command cAST;
6     public boolean hasReturn;
7
8     public FunctionDeclaration(TypeDenoter tdAST, Identifier iAST, FormalParameterSequence
          fpsAST, Command cAST) {
9        super();
10       this.tdAST = tdAST;
11       this.iAST = iAST;
12       this.fpsAST = fpsAST;
13       this.cAST = cAST;
14       hasReturn = false;
15    }
16
17    public FunctionDeclaration(TypeDenoter tdAST, Identifier iAST, FormalParameterSequence
          fpsAST) {
18       this(tdAST, iAST, fpsAST, null);
19       forwardDecl = true;
20    }
21
22    public Object visit(Visitor v, Object arg) {
23       return v.visitFunctionDeclaration(this, arg);
24    }
25 }
```

Listing 3.2: The `FunctionDeclaration` class.

The `FunctionDeclaration` class has a `visit` method, which is used in connection with the *visitor pattern* explained in Section 3.2.4.

3.2.4 The Visitor Pattern

The object-oriented design pattern known as the visitor pattern is used by the contextual analyzer and the code generator as the method for traversing the AST. See Figure 3.7 for an illustration of the visitor pattern related to the AST.

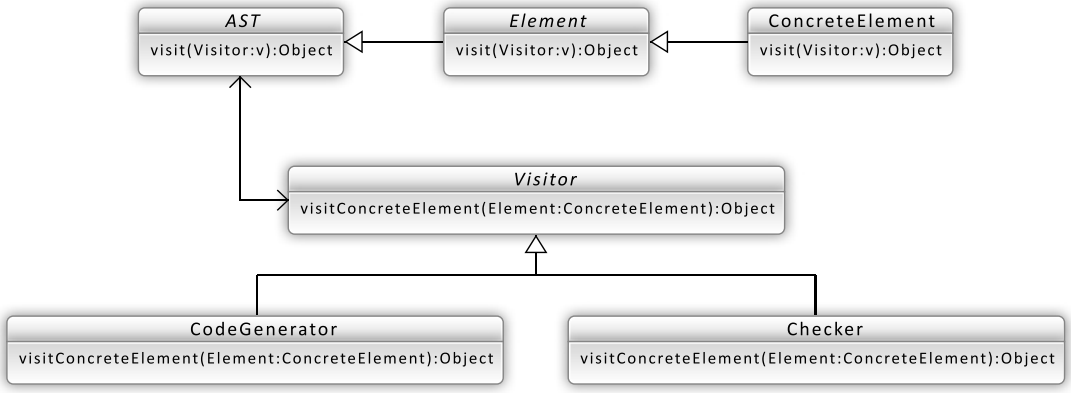

Figure 3.7: UML diagram of the visitor design pattern.

The concept of the visitor pattern related to the AST is as follows. As explained in Section 3.2.3 on page 25, all `AST` objects inherit from the abstract `AST` superclass, thus all `AST` objects will have a `visit` method. For all concrete classes this method takes a `Visitor` object as an argument and uses the corresponding visitConcreteElement method of the `Visitor` object with the `AST` object itself as an argument.

The visitConcreteElement method is as substitute for all the methods declared in the `Visitor` interface. A visitConcreteElement method exists for each concrete `AST` subclass, such that a method called `visitProgram` exists for the `Program` class, a method called `visitVariableDeclaration` exists for the `VariableDeclaration` class, etc. This makes it possible for any `Visitor` object to traverse all `AST` objects in the AST.

3.3 Syntactic Analysis

The first phase of the compilation process from KAL to KIL is syntactic analysis. The syntactic analyzer reads through the input KAL program and checks if it conforms with the syntax specification of KAL (described in Section 2.2 on page 6). The syntactic analyzer consists of a *scanner* (see Section 3.3.1) and a *parser* (see Section 3.3.2 on page 29). The scanner and the parser are both used to read source code, though the difference between the two can be expressed as the level of detail with which they operate. Where the scanner operates with characters and generates *tokens* (see Section 3.3.1), the parser operates with tokens and generates an AST. The result of successfully running the syntactic analyzer is thus the source program represented in the form of an AST.

3.3.1 Scanner

The purpose of the scanner is to read characters from the KAL program and return tokens. The scanner runs alongside the parser, meaning that it will not immediately scan the entire program, but instead return a single token when the appropriate method is called. The scanner consists of the methods mentioned in Table 3.2 on the next page, but only relevant parts of selected methods are shown in this section.

Tokens

Tokens are implemented as the `Token` class and consists of only two variables: `kind` and `spelling`. The kinds a token can have are categorized into identifiers, literals, operators, type denoters, punctuations, brackets, reserved words, and special tokens. For example, the kind of a token would be `IDENTIFIER` with the spelling “fooBar” if a variable named “fooBar” was declared in the source code of the program.

When a token object is instantiated and its kind is set to be either identifier or operator its spelling is matched against reserved words, such as “array” and “import”, and the token will get the corresponding kind depending on its spelling. The implementation of the `Token` class can be seen in Listing 3.3 on the following page. For the sake of simplicity, only three kinds are shown as an example.

Method	Returns	Description
Scanner(SourceFile sourceFile)	n/a	Constructor, loads the source code file and initiates the first character.
takeIt()	void	Appends the current character to currentSpelling and stores the next character in currentChar.
take(char expectedChar)	void	Same as takeIt(), but only if the current character equals expectedChar.
skipIt()	void	Skips the current character and stores the next character in currentChar.
skip(char expectedChar)	void	Same as skipIt(), but only if the current character equals expectedChar.
isLetter(char c)	boolean	Checks if a character is a letter (as defined by the lexicon).
isDigit(char c)	boolean	Checks if a character is a digit (as defined by the lexicon).
scanToken()	Token.Kind	Scans the source code file for non-separator characters (as defined by the lexicon) and returns a token kind.
scanSeparator()	void	Scans the source code file for separator characters (as defined by the lexicon) and ignores them.
scan()	Token	Determines how to scan the source code file, scans, and returns a Token object to the parser.

Table 3.2: Scanner class methods.

```
1  public class Token {
2     public Kind kind;
3     public String spelling;
4
5     public Token (Kind kind, String spelling) {
6        this.kind = kind;
7        this.spelling = spelling;
8        if(kind.equals(Kind.IDENTIFIER) || kind.equals(Kind.OPERATOR)) {
9           for(Kind k : Kind.values()) {
10             if(spelling.equals(k.toString())) {
11                this.kind = k;
12             }
13          }
14       }
15    }
16
17    public enum Kind {
18       ELSE,
19       FALSE,
20       FUNCTION;
21
22       public String toString() {
23          switch(this) {
24             case ELSE: return "else";
25             case FALSE: return "false";
26             case FUNCTION: return "function";
27             default: return "eot";
28          }
29       }
30    };
31 }
```

Listing 3.3: The Token class.

The Scanning Process

When a scanner object has been instantiated with a source code file, the scan method can be called. The process from calling the scan method to receiving a Token object is as follows. First, the file is scanned via the scanSeparator method for separator characters until a non-separator character appears. The scanToken method is then called, and will scan characters until they no longer match the rules stated by the lexicon. The token rules from the lexicon are implemented in the scanToken method, as can be seen in Listing 3.4, which shows the implementation of the rule *Identifier → Letter (Letter | Digit)**.

```
1  private Token.Kind scanToken() throws CompilationErrorException {
2     switch(Character.toLowerCase(currentChar)) {
3        // Identifier -> Letter ( Letter | Digit )*
4        case 'a': case 'b': case 'c': /*...*/ case 'z':
5           takeIt();
6           while(isLetter(currentChar) || isDigit(currentChar)) {
7              takeIt();
8           }
9
10          ... // Code related to libraries
11
12          return Token.Kind.IDENTIFIER;
13
14       .
15       .
16       .
17    }
18 }
```

Listing 3.4: The scanToken method.

First, a check for whether the current character is a letter is performed. If the character indeed is a letter, the character is appended to the variable currentSpelling. The scanToken method then proceeds to scan characters for either letters or digits, and append matching characters to the variable currentSpelling. When the current character no longer is either a letter or a digit the token kind is returned to the scan method.

The scanner object now has the data needed to create a token, which is stored in the variables currentSpelling and currentKind, and a new Token object is created and returned to the parser from the scan method.

3.3.2 Parser

The purpose of the parser is to decide whether or not the source code provided for the scanner is a correct expression of the language KAL. This is done by scanning the source code, analyzing its phrase structure with the parser, and representing the structure in an AST. The parser uses the scanner to read the source code file, where the parser operates with the tokens it has been given by the scanner.

A complete stream of tokens will represent a complete phrase, and the parser is then able to determine if the phrase is correctly structured depending on the grammar rules defined in the EBNF. The parser does not determine whether the phrase is contextually correct, but only that

the syntax is correct. For example, the parser accepts source code in which a variable is used but not declared since it is syntactically correct.

The KAL parser uses the recursive-descent algorithm, which is a top-down approach to parsing. This means that the parser analyzes the source code from the root first and then determines the phrase structure by the child elements with the use of tokens. The parser has a parse method for each nonterminal symbol in the grammar, which is nested in a way suited for creating a tree structured program in the form of an AST. As an example, observe the first two rules of the EBNF:

(1)	Program	→	Import* Declaration Declaration*
(2)	Import	→	**import** Identifier **;**

These rules are implemented in the `parseProgram` method as shown in Listing 3.5.

```
1  public Program parseProgram() {
2     Program prog = new Program();
3     try {
4        while (currentToken.kind != Token.Kind.EOT) {
5           switch (currentToken.kind) {
6           case IMPORT:
7              acceptIt();
8              String filename = currentToken.spelling;
9              libraryFilenames.add(filename + ".");
10             acceptIt();
11             scanner.sourceFile.importLibrary(filename + ".kll");
12             accept(Token.Kind.SEMICOLON);
13             break;
14          default:
15             prog.dASTList.add(parseDeclaration());
16
17             if (libraryFilenames.size() > scanner.sourceFile.libraryLevel + 1) {
18                int last = libraryFilenames.size() - 1;
19                libraryFilenames.remove(last);
20             }
21
22             break;
23          }
24       }
25    }
26    catch(CompilationErrorException e) {
27       System.out.println(errorReport.errorDescription);
28    }
29
30    return prog;
31 }
```

Listing 3.5: The `parseProgram` method.

The first rule states that a Program consists of none or any amount of Imports, followed by at least one Declaration. In the `parseProgram` method this is implemented as shown in Listing 3.5. A while loop evaluates the kind of all tokens until having reached the last token (EOT kind), and this is what makes the stream of tokens.

Only Imports or Declarations are expected, which is implemented as a switch which checks the kind of the current token. If the kind of the current token is `IMPORT`, the corresponding block of code is executed, otherwise the switch defaults to the code block for Declarations. If the current token kind is not an Import or a Declaration the source code does not obey the first rule of the EBNF, and a parsing error will be reported.

If no mismatches occur the parser will proceed parsing in the downwards direction, and as such it calls the `parseDeclaration` method for each Declaration kind of token it is presented with. The rules 3 to 8 in the EBNF concern the parsing of declarations. As an example, observe the rules 3 and 4 of the EBNF:

---

(3)	Declaration	→	VariableDeclaration **;**
		|	ConstantDeclaration **;**
		|	ArrayDeclaration **;**
		|	ProcedureDeclaration
		|	FunctionDeclaration
(4)	VariableDeclaration	→	TypeDenoter Identifier

It is stated by rule 3 that a Declaration can be a VariableDeclaration. It is shown in Listing 3.6 how VariableDeclarations are recognized.

```
1  private Declaration parseDeclaration() throws CompilationErrorException {
2     Declaration declAST = null;
3
4     switch (currentToken.kind) {
5        case INT:
6        case BOOL:
7        case CHAR:
8        case FLOAT:
9           declAST = parseVariableDeclaration();
10          accept(Token.Kind.SEMICOLON);
11          break;
12
13       ... // Cases for constants, arrays, functions and procedures
14
15       default:
16          errorReport = new ErrorReport("Type denoter, keyword 'constant', keyword
                'array', keyword 'function' or keyword 'procedure'", currentToken);
17    }
18    return declAST;
19 }
```

Listing 3.6: The `parseDeclaration` method.

As rule 4 states, a VariableDeclaration consists of a TypeDenoter followed by an Identifier. A TypeDenoter can be either an integer, a boolean, a character or a float. If the first token of the Declaration is indeed a TypeDenoter the Declaration is expected to be a VariableDeclaration and conforming to the 4th rule of the EBNF. The `parseVariableDeclaration` method is then called, and is shown in Listing 3.7 on the following page.

```
1  private Declaration parseVariableDeclaration() throws CompilationErrorException {
2     Declaration declAST;
3
4     TypeDenoter typeAST = parseTypeDenoter();
5     Identifier idAST = parseIdentifier();
6
7     declAST = new VariableDeclaration(typeAST, idAST);
8
9     return declAST;
10 }
```

Listing 3.7: The `parseVariableDeclaration` method.

In the `parseVariableDeclaration` method two other methods are called: `parseTypeDenoter` and `parseIdentifier`. Both set leaf nodes for the AST. An example of how `parseTypeDenoter` sets a BoolTypeDenoter as a leaf node is shown in Listing 3.8.

```
1  private TypeDenoter parseTypeDenoter() throws CompilationErrorException {
2     TypeDenoter typeAST = null;
3     switch (currentToken.kind) {
4        case BOOL:
5           typeAST = new BoolTypeDenoter();
6           acceptIt();
7           break;
8
9        ... // Cases for char, float and int
10
11       default:
12          errorReport = new ErrorReport("bool, char, float or int", currentToken);
13    }
14    return typeAST;
15 }
```

Listing 3.8: The `parseTypeDenoter` method.

As can be seen, no more calls to parse methods are performed. When the leaf nodes are generated the current VariableDeclaration branch is complete and the `parseProgram` method will then proceed onto the next Declaration. As an example, the AST for the source code `bool SomeBoolean;` is illustrated in Figure 3.8 and the steps for creating the AST from tokens returned from a scan of the source code are explained in Listing 3.9 on the next page.

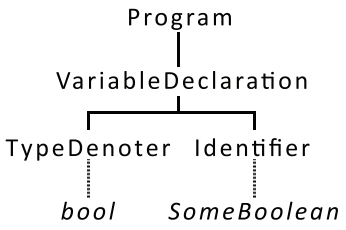

Figure 3.8: AST for the source code `bool SomeBoolean;`

```
Current token stream: <Bool><Identifier><Semicolon><EOT>
Method called: parseProgram
   Current token kind is not EOT, thus proceed
   Current token kind is not Import, thus a Declaration is expected
   Method called: parseDeclaration
      Current token kind is Bool, thus a VariableDeclaration is expected
      Method called: parseVariableDeclaration
         TypeDenoter expected
         Method called: parseTypeDenoter
            <Bool> accepted
            Current token stream: <Identifier><Semicolon><EOT>
            Returned to method parseVariableDeclaration
         Identifier expected
         Method called: parseIdentifier
            <Identifier> accepted
            Current token stream: <Semicolon><EOT>
            Returned to method parseVariableDeclaration
         Returned to method parseDeclaration
         <Semicolon> accepted
         Current token stream: <EOT>
      Returned to method parseProgram
   Current token kind is EOT, thus finish
AST created
```

Listing 3.9: Example steps for creating an AST from tokens.

The example steps shown in Listing 3.9 illustrates exactly how the EBNF rules are used to traverse the tokens generated from the source code, resulting in the example AST shown in Figure 3.8 on the preceding page. The indentations in the text corresponds to the node levels in the AST, and it shows how the EBNF is traversed like a tree going from the root (the `parseProgram` method) to the leaves (the `parseTypeDenoter` and `parseIdentifier` methods) and back towards the root to check for more branches.

3.4 Contextual Analysis

When the syntactic analysis is complete a `Checker` object is instantiated, which receives an AST from the parser. The purpose of the `Checker` is to traverse the AST (see Section 3.2.3 on page 25) and check that the KAL code contains no scope or type errors.

The `Checker` class consists of a number of methods which are defined in the `Visitor` interface (see Section 3.2.4 on page 26). These methods are used to traverse the AST. Listing 3.10 shows the constructor of the `Checker`.

```
1 public Checker() {
2    idTable = new IdentificationTable();
3    establishStdEnvironment();
4 }
```

Listing 3.10: `Checker` class constructor.

When the `Checker` class is instantiated the constructor will instantiate the `IdentificationTable` class, which will be explained in more detail in Section 3.4.1 on page 35.

After the IdentificationTable has been instantiated the establishStdEnvironment method will be called. The method declares the standard constants, routines and operators in KAL. This means that the given constants, routines and operators do not have to be declared by the programmer before use. In Listing 3.11 some examples of the standard declarations can be seen. First, the different standard types of KAL are instantiated along with the anyType type. The anyType type is used for some of the standard functions in KAL that take any of the types as a parameter or returns a different type depending on the given parameter.

```
1  private void establishStdEnvironment() {
2     StdEnvironment.integerType = new IntTypeDenoter();
3     ... // Rest of primitive types
4     StdEnvironment.anyType = new AnyTypeDenoter();
5
6     StdEnvironment.trueDecl = declareStdConstant("true", StdEnvironment.booleanType, new
          BooleanExpression(new BooleanLiteral("true")));
7     StdEnvironment.falseDecl = declareStdConstant("false", StdEnvironment.booleanType, new
          IntegerExpression(new IntegerLiteral("false")));
8
9     StdEnvironment.notDecl = declareStdUnaryOperator("!", StdEnvironment.booleanType,
          StdEnvironment.booleanType);
10    ... // Rest of operators
11
12    StdEnvironment.runDecl = declareStdProcedure("Run", new FormalParameterSequence(),
          true);
13
14    StdEnvironment.toInt = declareStdFunction("ToInt", StdEnvironment.integerType,
          initFPS(StdEnvironment.anyType));
15    ... // Rest of primitive routines
16 }
```

Listing 3.11: The establishStdEnvironment method.

After the types have been declared the constants true and false are declared. Then the unary operator not (!) is declared with a spelling, an argument type, and a result type. The same goes for the binary operators though they have a second argument before the result type.

After this, the Run procedure is declared with an empty FormalParameterSequence meaning that it has no parameters. It is also declared as a forward declaration meaning that this procedure has to be implemented in the KAL source program. The third parameter of declareStdProcedure method is set to true to indicate that it is the Run procedure.

The primitive routines (see Table 3.5 on page 45) are declared with a FormalParameterSequence describing the parameters as well as a return type for primitive routines that return a value.

To initiate the tree traversal the check method must be called. The check method has one parameter, which is a Program AST. It starts the traversal of the AST by calling the visit method of the Program with the Checker and null as arguments.

3.4.1 Scope Checking

One part of the contextual analysis is checking that the KAL code comply with the scope rules of KAL presented in Section 2.3.1 on page 11. In KAL this is done by using the IdentificationTable class containing objects of the Element class and as well as auxiliary methods for adding and retrieving elements and opening and closing scopes.

Element

An Element object consists of three variables describing an identifier:

- An integer representing the scope level of the identifier.
- A string representing the spelling of the identifier.
- The declaration AST corresponding to the identifier.

In Listing 3.12 the constructor of the Element class can be seen.

```
1 public Element(int scopeLevel, String id, Declaration decl) {
2    this.setScopeLevel(scopeLevel);
3    this.setId(id);
4    this.setDecl(decl);
5 }
```

Listing 3.12: Element class constructor.

IdentificationTable

The IdentificationTable contains a variable scopeLevel holding the current scope level and a list of Element objects representing identifiers in the KAL code. The constructor of the IdentificationTable can be seen in Listing 3.13. This constructor sets the value of scopeLevel to 0 since all programs start in the global scope.

```
1 public IdentificationTable() {
2    elements = new ArrayList<Element>();
3    scopeLevel = 0;
4 }
```

Listing 3.13: IdentificationTable class constructor.

The enter method shown in Listing 3.14 on the next page is called for every declaration in the KAL code. Here the list containing the previously declared identifiers will be checked for duplicates. This check is performed by checking if the spelling of two identifiers are the same. A special case occurs when a function is declared and entered since it is then necessary to check if it is a forward declaration. The following are possible outcomes of the check:

- If the entered function (forward declaration or not) is the only one with that spelling, it is entered in the table.
- There are two possible outcomes if a declaration with the same spelling already exists:
  - If the existing declaration is a forward declaration and the new declaration is not, the forward declaration is removed and the new declaration is entered.

– If the existing declaration is not a forward declaration or both declarations are forward declarations, an error will occur later in the process.

```
1  public void enter(String id, Declaration decl) {
2     for (int i = elements.size() - 1; i > 0; i--) {
3        if (elements.get(i).getId().equals(id)) {
4           if (elements.get(i).getDecl().forwardDecl == true && !decl.forwardDecl) {
5              elements.remove(i);
6           } else {
7              decl.duplicateDecl = true;
8           }
9        }
10    }
11    elements.add(new Element(scopeLevel, id, decl));
12 }
```

Listing 3.14: The `enter` method.

In the `IdentificationTable` there are two auxiliary methods, which are used to keep track of the scope level when the KAL code is being checked. The first is the `openScope` method and the other is the `closeScope` method.

The `openScope` method seen in Listing 3.15 increases the variable `scopeLevel` by one.

```
1  public void openScope() {
2     scopeLevel++;
3  }
```

Listing 3.15: The `openScope` method.

The method `closeScope` seen in Listing 3.16 decreases the scope level and makes sure that all variables in the current scope level are removed from the `IdentificationTable`. This way locally declared variables in different local scope levels do not affect each other. After the removal of these variables the scope level is decreased by one.

```
1  public void closeScope () {
2     for (int i = elements.size() - 1; i > 0; i--) {
3        if (elements.get(i).getScopeLevel() == scopeLevel) {
4           elements.remove(i);
5        } else {
6           break;
7        }
8     }
9     scopeLevel--;
10 }
```

Listing 3.16: The `closeScope` method.

As declarations are entered in the `IdentificationTable` a way to retrieve the information is needed every time a variable is used in a KAL program. The method for retrieving a declaration can be seen in Listing 3.17 on the facing page. This method enables type checking of variables

and functions along with giving a way of checking if a variable has been declared. The method searches the list of `Element` objects to see if there is an `Element` object with a spelling matching the parameter. If this is the case then the declaration of this identifier is returned. If there is no `Element` object in the list with that spelling then null is returned to indicate that the identifier has not been declared.

```
1 public Declaration retrieve(String spelling) {
2    for (int i = elements.size() - 1; i >= 0; i--) {
3       if (elements.get(i).getId().equals(spelling)) {
4          return elements.get(i).getDecl();
5       }
6    }
7    return null;
8 }
```

Listing 3.17: The `retrieve` method.

The last auxiliary method in the `IdentificationTable` is a method used to get the return type of a function and make sure that a function includes a return command. The method searches backwards through the list of `Element` objects for the last function declared. Then a flag is set to indicate that the function has a return command and then returns the type of that function.

```
1  public TypeDenoter getReturnType() {
2     for (int i = elements.size() - 1; i >= 0; i--) {
3        Declaration decl = elements.get(i).getDecl();
4        if (decl instanceof FunctionDeclaration) {
5           FunctionDeclaration func = (FunctionDeclaration) decl;
6           func.hasReturn = true;
7           return func.tdAST;
8        }
9     }
10    return null;
11 }
```

Listing 3.18: The `getReturnType` method.

Checker

The scope check is performed in two steps. The first step is done by the methods `visitVariableDeclaration`, `visitFunctionDeclaration` and `visitProcedureDeclaration`. The `visitVariableDeclaration` can be seen in Listing 3.19 on the next page. This method uses the `enter` method from the `IdentificationTable` to check that the newly declared variable complies with the scope rules stated in Section 2.3.1 on page 11. If the `enter` method flags the declaration as a duplicate the `visitVariableDeclaration` reports an error.

The method `visitFunctionDeclaration` shown in Listing 3.20 on the next page is implemented similarly. It calls the `enter` method, which will perform the first scope checking as before with the `visitVariableDeclaration`. After this the `openScope` method is called to increase the scope level. This ensures that all variables declared in this scope are at a higher scope level.

```
1 public Object visitVariableDeclaration(VariableDeclaration decl, Object arg) {
2    decl.tAST.visit(this, arg);
3    idTable.enter(decl.iAST.spelling, decl);
4    if (decl.duplicateDecl) {
5       System.out.println("Variable \"" + decl.iAST.spelling + "\" is a duplicate");
6       errorReport.error = true;
7    }
8    return null;
9 }
```

Listing 3.19: The `visitVariableDeclaration` method.

A check is performed to ensure that the function is not a duplicate. If the function is not a forward declaration the method also ensures that a return command exists at the scope level of the function. If a function is flagged as having no return command by the `getReturnType` method an error is reported. Lastly, the `closeScope` method is called. The `visitProcedure-Declaration` method is implemented similarly.

```
1  public Object visitFunctionDeclaration(FunctionDeclaration decl, Object arg) {
2     ... // Perform type check.
3
4     idTable.enter(decl.iAST.spelling, decl);
5     idTable.openScope();
6
7     ... // Visit the return type, identifier and formal parameter sequence.
8
9     if (decl.duplicateDecl) {
10       System.out.println("Function \"" + decl.iAST.spelling + "\" is a duplicate!");
11       errorReport.error = true;
12    }
13    if (decl.cAST != null) {
14
15       ... // Visit the commands of the function.
16
17       if (!decl.hasReturn) {
18          System.out.println("The function \"" + decl.iAST.spelling + "\" has no RETURN!");
19          errorReport.error = true;
20       }
21    }
22    idTable.closeScope();
23    return null;
24 }
```

Listing 3.20: The `visitFunctionDeclaration` method.

The second step of the scope checking is performed at an applied occurrence of a variable or routine. The corresponding identifier will be visited by the `visitIdentifier` method shown in Listing 3.21 on the facing page to ensure that it has been declared. To make this check the auxiliary method `retrieve` from the `IdentificationTable` is called with the spelling of the identifier. If the return value of the method is null it means that the variable or routine has not been declared resulting in an error.

```
1 public Object visitIdentifier(Identifier id, Object arg) {
2    id.decl = (Declaration) idTable.retrieve(id.spelling);
3    if (id.decl == null) {
4       System.out.println(id.spelling + " is not declared!");
5       errorReport.error = true;
6    }
7    return id.decl;
8 }
```

Listing 3.21: The `visitIdentifier` method.

3.4.2 Type Checking

The second part of the contextual analysis is type checking. With KAL this is done by checking if the code complies with the type rules of KAL. This is done in the `Checker` class alongside the scope checking. Only relevant parts are covered in this section.

Listing 3.22 shows how the part of the `visitFunctionDeclaration` method regarding forward declarations has been implemented. The method checks if the forward declaration is of type `FunctionDeclaration`. If this is the case the method will check if the return type of the function and the forward declaration match. After that it will call the method `matchFormalParameters`, which will be explained later. If the forward declaration is not of type `FunctionDeclaration` but of type `ProcedureDeclaration` an error will be reported, since a procedure cannot be declared as a function.

```
1  public Object visitFunctionDeclaration(FunctionDeclaration decl, Object arg) {
2     Declaration forwDecl = (Declaration) idTable.retrieve(decl.iAST.spelling);
3     if (forwDecl instanceof FunctionDeclaration) {
4        FunctionDeclaration forwFuncDecl = (FunctionDeclaration) forwDecl;
5
6        if (!forwFuncDecl.tdAST.equals(decl.tdAST)) {
7           System.out.println("Forward decl and decl have different return type!");
8           errorReport.error = true;
9        }
10       matchFormalParameters(forwFuncDecl.fpsAST.dASTList, decl.fpsAST.dASTList);
11    } else if (forwDecl instanceof ProcedureDeclaration){
12       System.out.println("The function \"" + decl.iAST.spelling + "\" has been forward
              declared as a procedure!");
13       errorReport.error = true;
14    }
15    .
16    .
17    .
18    // Rest of type checking
```

Listing 3.22: The `visitFunctionDeclaration` method.

Listing 3.23 on the following page shows how the `matchFormalParameters` method has been implemented. This method compares the parameters of a routine declaration with the parameters of the corresponding forward declaration. The method starts by checking the size of the two parameter lists, which must be the same. Afterwards, the lists is run through one element at a time to check if the types of the elements match. The type of an element can be either

`VariableDeclaration` or `ArrayDeclaration`. If the types of the elements do not match, an error is reported. After this, the types of the declarations in the KAL code are checked. If they do not match, an error is reported.

```
1  public void matchFormalParameters(ArrayList<Declaration> forwDeclList,
       ArrayList<Declaration> declList) {
2     if (forwDeclList.size() == declList.size()) {
3        for (int i = 0; i < forwDeclList.size(); i++) {
4           if (forwDeclList.get(i) instanceof VariableDeclaration && declList.get(i)
                 instanceof VariableDeclaration) {
5              VariableDeclaration
6              varDecl = (VariableDeclaration) forwDeclList.get(i),
7              varDecl2 = (VariableDeclaration) declList.get(i);
8              if (!varDecl.tAST.equals(varDecl2.tAST)) {
9                 System.out.println("Type mismatch in formalparametersequence!"
10                      + "\nForward declaration type: " + varDecl.tAST
11                      + "\nDeclaration type: " + varDecl2.tAST);
12                errorReport.error = true;
13             }
14          } else if (forwDeclList.get(i) instanceof ArrayFormalDeclaration &&
                 declList.get(i) instanceof ArrayFormalDeclaration) {
15             ArrayFormalDeclaration
16             arrayDecl = (ArrayFormalDeclaration) forwDeclList.get(i),
17             arrayDecl2 = (ArrayFormalDeclaration) declList.get(i);
18             if (!arrayDecl.tdAST.equals(arrayDecl2.tdAST)) {
19                System.out.println("Type mismatch in formalparametersequence!"
20                      + "\nForward declaration type: " + arrayDecl.tdAST
21                      + "\nDeclaration type: " + arrayDecl2.tdAST);
22                errorReport.error = true;
23             }
24          } else {
25             System.out.println("Type mismatch in formalparametersequence!"
26                   + "\nForward declaration is of type: " + forwDeclList.get(i)
27                   + "\nDeclaration is of type: " + declList.get(i));
28             errorReport.error = true;
29          }
30       }
31    } else {
32       System.out.println("FormalParameters does not match!!"
33             + "\nForward declaration has: " + forwDeclList.size() + " parameters"
34             + "\nDeclaration has: " + declList.size() + " parameters");
35       errorReport.error = true;
36    }
37 }
```

Listing 3.23: The `matchFormalParameters` method.

Listing 3.24 shows how the `visitBinaryExpression` method has been implemented. It starts by visiting the two expressions and the operator. If the operator is not a binary operator the checker reports an error. After that it checks if the type of the first argument of the operator is of anyType. If it is, a check is performed to ensure that the types of the two expressions are the same, otherwise an error is reported. If the first argument of the operator is not of type anyType it checks if the types of the expressions match the types expected by the arguments of the operator. Finally, the method checks the result type of the operator. If it is of type anyType, the type of the first expression is returned, otherwise the result type of the operator is returned.

```
1  public Object visitBinaryExpression(BinaryExpression exp, Object arg) {
2     TypeDenoter exp1Type = (TypeDenoter) exp.e1AST.visit(this, null);
3     TypeDenoter exp2Type = (TypeDenoter) exp.e2AST.visit(this, null);
4     Declaration op = (Declaration) exp.oAST.visit(this, null);
5
```

```
6     if (! (op instanceof BinaryOperatorDeclaration)) {
7        System.out.println("Operator is not a binary operator");
8        errorReport.error = true;
9     }
10
11    BinaryOperatorDeclaration op2 = (BinaryOperatorDeclaration) op;
12
13    if (op2.arg1Type == StdEnvironment.anyType) {
14       if (! (exp1Type.equals(exp2Type))) {
15          System.out.println("Binary expression consists of two different types");
16          errorReport.error = true;
17       }
18    }
19    else if (! (exp1Type.equals(op2.arg1Type))) {
20       System.out.println("Wrong type for argument 1");
21       errorReport.error = true;
22    }
23    else if (! (exp2Type.equals(op2.arg2Type))) {
24       System.out.println("Wrong type for argument 2");
25       errorReport.error = true;
26    }
27
28    if (op2.resultType == StdEnvironment.anyType) {
29       exp.type = exp1Type;
30    } else {
31       exp.type = op2.resultType;
32    }
33    return exp.type;
34 }
```

Listing 3.24: The `visitBinaryExpression` method.

Listing 3.25 shows how the `visitAssignCommand` method has been implemented. The method starts by checking if the identifier of the command is of type `VariableDeclaration`. If it is not an error is reported. Afterwards, it checks if the expression and the identifier are of the same type, otherwise an error is reported.

```
1  public Object visitAssignCommand(AssignCommand cmd, Object arg) {
2     Declaration decl = (Declaration) cmd.iAST.visit(this, arg);
3
4     if (decl instanceof VariableDeclaration) {
5        VariableDeclaration var = (VariableDeclaration) decl;
6        TypeDenoter vType = var.tAST;
7        TypeDenoter eType = (TypeDenoter) cmd.eAST.visit(this, arg);
8
9        if (! (vType.equals(eType)) && eType != null) {
10          System.out.println("Expression type is different from Variable type");
11          errorReport.error = true;
12       }
13    } else if (decl instanceof ArrayDeclaration) {
14       System.out.println("\"" + cmd.iAST.spelling + "\" is an array!");
15    }
16    .
17    .
18    .
19    else {
20       System.out.println("\"" + cmd.iAST.spelling + "\" is not declared as a variable!");
21       errorReport.error = true;
22    }
23    return null;
24 }
```

Listing 3.25: The `visitAssignCommand` method.

3.5 KVM

In this section we introduce the *KAL Abstract Machine* (*KAM*), which is a theoretical machine meant to execute the instructions written in the intermediate language KIL, which the language KAL is compiled to. The purpose of an abstract machine is to provide a set of rules and specifications for a machine independent of implementation details.

Later in this section we describe the development of the KVM, which is the software implementation of KAM.

3.5.1 Abstract Machine Description

KAM is an abstract machine capable of executing simple instructions and performing primitive arithmetic and logical operations. All evaluation and memory allocation is done on a stack, but registers are used to keep track of certain variables and for calling *primitive routines*. In KAM there are two different storages:

Code store

The code store in KAM is segmented into two parts. The first part is the *code segment*, which contains the instructions that the machine is to execute. The second part is the *primitives segment*, which is where the machine's primitive routines are defined. Primitive routines are elementary arithmetic and logical operations as well as operations for handling input/output and some run-time error handling. The boundaries of the segments in the code store are constant and specified by their respective registers.

Data store

The data store is not segmented and is used exclusively for the machine's stack, but the stack itself is fragmented. The bottom segment of the stack is the *global segment*, which contains all the data which is accessible from anywhere in the program. The rest of the stack can be segmented into any number of smaller local segments which are called *frames*. A frame is pushed on to the stack every time a routine is called. The frame contains the local data for the routine as well as a link to the underlying frame and a return address. When the program returns from a routine, the frame is popped and the program continues running from the frame's return address.

As the machine pushes and pops values and addresses from the stack during run-time, the boundaries of the stack change which is reflected in the stack registers.

As the data store can contain both addresses and values, which can be either integers or floating points, each entry in the stack will be referred to as an object for easier readability, unless differentiation is required by the context.

Registers

KAM contains a number of registers which it uses for various operations. Table 3.3 on the next page shows a list of registers in KAM. All of the registers in KAM are only to be used for their

respective purposes meaning that there are no general-purpose registers in KAM.

Register Number	Register Mnemonic	Register Name	Function
0	CB	Code base	Constant. Specifies where the bottom of the code segment in the code store is located.
1	CT	Code top	Constant. Specifies where the top of the code segment in the code store is located.
2	PB	Primitives base	Constant. Specifies where the bottom of the primitives segment in the code store is located.
3	PT	Primitives top	Constant. Specifies where the top of the primitives segment in the code store is located.
4	SB	Stack base	Constant. Specifies where the bottom of the stack is located.
5	ST	Stack top	Variable. Specifies where the top of the stack is located. This value is incremented every time the machine pushes an object to the stack, and decremented every time it pops one.
6	LB	Local base	Variable. Specifies where the current local base is located in the stack. This value changes whenever a routine is called.
7	CP	Code pointer	Variable. Specifies which instruction in the code segment in the code store is to be executed next. Changes after each instruction.

Table 3.3: KAM registers.

Instructions

The KAM machine can execute the 13 instructions shown in Table 3.4 on the following page. Each instruction consists of 4 fields in the following order:

- Operation code (*op*)
- Operand size (*n*)
- Address displacement (*d*)
- Register number (*r*)

The op-field is used to determine which operation the machine is to perform. Together the d-field and r-field make up the operand's address which is used to determine where to either store an object or fetch an object from. The *n* field describes the size of the operand. Not all fields are required in each instruction and some may therefore be left blank. When describing KAM instructions in Table 3.4 on the next page, we differentiate between addresses referring to the data store and those referring to the code store by calling them *data addresses* and *code addresses* respectively.

Op-code	Instruction Mnemonic	Function
0	`LOAD n d r`	Fetch *n* objects from the data address (*d* + register *r*) and push them on to the stack.
1	`LOADA d r`	Push the data address (*d* + register *r*) on to the stack.
2	`LOADI n`	Pop a data address from the stack, fetch *n* objects from that address, and push them on to the stack.
3	`LOADL d`	Push a literal value *d* onto the stack.
4	`STORE n d r`	Pop *n* objects from the stack, and store them at the data address (*d* + register *r*).
5	`STOREI n`	Pop a data address from the stack, then pop *n* objects from the stack, and store them at that address.
6	`CALL d r`	Call the routine at code address (*d* + register *r*).
7	`RETURN n d`	Return from the current routine: pop *n* objects from the stack as the result, then pop the topmost frame, then pop *d* objects for the routine arguments, then push the result back on to the stack.
8	`PUSH d`	Push *d* objects (uninitialized) on to the stack.
9	`POP n d`	Pop *n* objects from the stack as the result, the pop *d* more words, then push the result back on to the stack.
10	`JUMP d r`	Jump to code address (*d* + register *r*).
11	`JUMPIF n d r`	Pop a value from the stack, then jump to code address (*d* + register *r*) if and only if that value equals *n*.
12	`HALT`	Stops the execution of the program.

Table 3.4: KAM instructions.

Routines

Routines in KAM all follow the same execution pattern:

1. Load routine arguments on to the stack top.
2. Call and execute the routine.
3. Return from the routine and replace the arguments on the stack top with the routine result.

KAM supports two kinds of routines: primitive routines and *code routines*.

Primitive routines are implemented in the primitives section of the code store and their body of code is therefore not expressed in KAM instructions. They perform simple arithmetic operations such as addition, subtraction, multiplication and division, and boolean operations such as checking whether two values are equal or not. A list of all primitive routines and their functions are shown in Table 3.5 on the facing page where v denotes a value:

Code routines are routines which are expressed as KAM instructions in the code segment of the code store. The procedure for calling a code routine is illustrated in Figure 3.9 on page 46 as a 3-step process.

Address	Mnemonic	Arguments	Function
PB + 1	ADD	v_1, v_2	Returns $v_1 + v_2$ as the result.
PB + 2	SUB	v_1, v_2	Returns $v_1 - v_2$ as the result.
PB + 3	MULT	v_1, v_2	Returns $v_1 * v_2$ as the result.
PB + 4	DIV	v_1, v_2	Returns v_1 / v_2 as the result.
PB + 5	MOD	v_1, v_2	Returns $v_1 \% v_2$ as the result.
PB + 6	NOT	v	Read v as boolean (false if 0, otherwise true) and return $\neg v$.
PB + 7	NEG	v	Returns $-v$ as the result.
PB + 8	AND	v_1, v_2	Returns $v_1 \wedge v_2$ as a boolean result.
PB + 9	OR	v_1, v_2	Returns $v_1 \vee v_2$ as a boolean result.
PB + 10	LT	v_1, v_2	Returns $v_1 < v_2$ as a boolean result.
PB + 11	LE	v_1, v_2	Returns $v_1 \leq v_2$ as a boolean result.
PB + 12	GT	v_1, v_2	Returns $v_1 > v_2$ as a boolean result.
PB + 13	GE	v_1, v_2	Returns $v_1 \geq v_2$ as a boolean result.
PB + 14	EQ	v_1, v_2	Returns $v_1 = v_2$ as a boolean result.
PB + 15	NEQ	v_1, v_2	Returns $v_1 \neq v_2$ as a boolean result.
PB + 16	WRITECHAR	v	Read v as integer (must be in range 0-255, else throw run-time exception) and write the ASCII character v.
PB + 17	WRITEINT	v	Read v as integer and write v.
PB + 18	WRITEFLOAT	v	Read v as float and write v.
PB + 19	WRITEBOOL	v	Read v as boolean (false if 0, otherwise true) and write v.
PB + 20	READCHAR		Read a character from input and push it on to the stack.
PB + 21	READINT		Read an integer from input and push it on to the stack.
PB + 22	READFLOAT		Read a float from input and push it on to the stack.
PB + 23	READBOOL		Read a boolean from input and push it on to the stack.
PB + 24	TOINT	v	Convert v to an integer and return it as the result.
PB + 25	TOFLOAT	v	Convert v to a float and return it as the result.
PB + 26	TOCHAR	v	Convert v to an integer (must be in range 0-255, else throw run-time exception) and return it as the result.
PB + 27	CHECKINDEXBOUNDS	v_1, v_2	If $v_1 < 0$ or $v_1 >= v_2$ throw run-time exception, otherwise return v_1 as the result.
PB + 28	SIZEOF	v	Return the value stored at the address v as the result.

Table 3.5: Summary of KAM primitive routines.

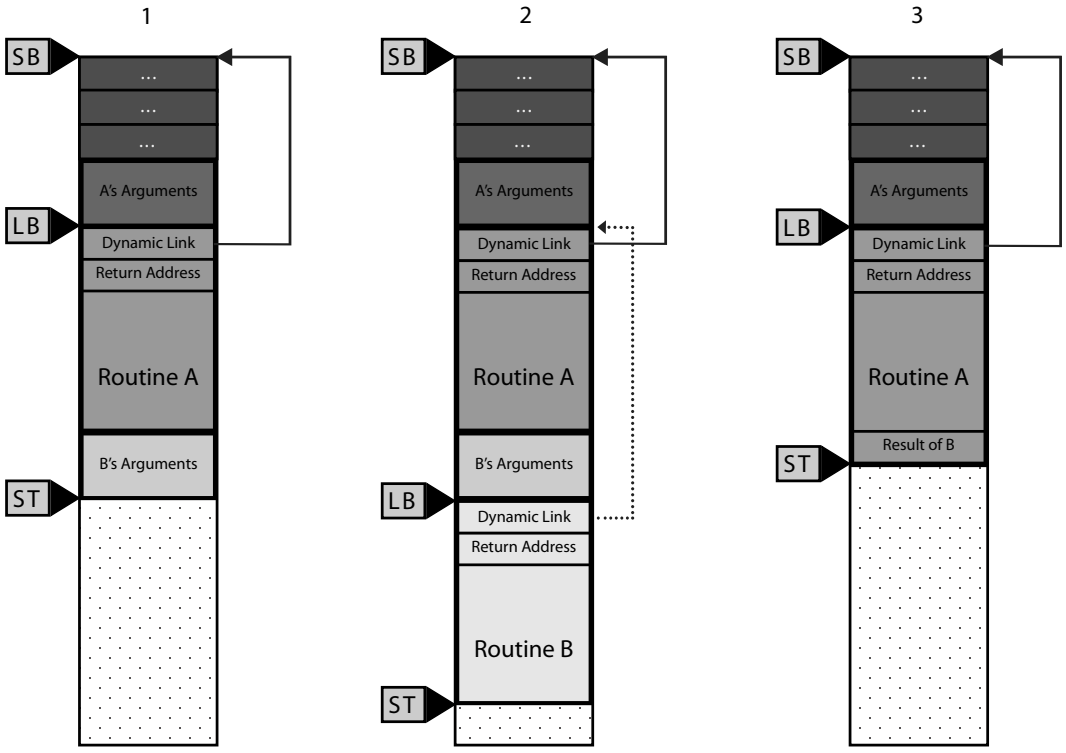

Figure 3.9: Calling a routine B from a routine A.

1. The arguments for the routine B are pushed onto the stack, and the `CALL` instruction is executed.

2. A new frame is pushed on to the stack for the routine B. The first two objects in the frame are the dynamic link and the return address. The dynamic link is an address to the bottom of A's frame and the return address points to where the routine was called from in the code store. Together these are called the routine's *link data*.

   The LB register is then changed to point to the base of B's frame and the CP register is changed to point to the beginning of the routine body in the code store. Each routine can access data on the stack relative to the SB and LB registers. The SB register gives the routine access to global data while the LB register gives it access to its local data. Because a routine's arguments are always pushed on to the stack before calling the routine, they can be accessed by a negative displacement relative to the LB register.

3. When the routine has been executed the result is pushed on to the stack and the `RETURN` instruction is executed, which pops the result, pops B's frame, pops it's arguments and then pushes the result back on to the stack. It also makes the CP register point to the B's return address and the LB register is changed to point to the address in the dynamic link.

3.5.2 Interpretation

When creating an abstract machine as a software implementation it is known as a *virtual machine*. An example of this is the Java Virtual Machine [LY99], which is described as the implementation of an abstract computing machine.

The KVM is implemented as an iterative interpreter, where an instruction in the code store is fetched, each of the instruction's fields are read and analyzed, after which the instruction is executed. This cycle repeats until the program either stops naturally or an error occurs which causes the program to halt.

The KVM consists of the following three classes: `Machine`, `Instruction` and `Interpreter`.

The `Machine` class is implemented as a static class in which the code store, data store and all the necessary information about the machine such as register addresses, instruction op-codes, primitive routine addresses and register contents are contained.

The code segment of the code store is implemented as an array of `Instruction` objects. The data store is implemented as an array of objects so that each entry in the array can hold either an `Integer` or a `Float` object.

The registers are simply implemented as integer variables as they only need to hold integer values.

The register addresses are implemented as an enumeration `RegisterAddress`. This is done to make it easier to differentiate between references to the contents of registers and references to their addresses. Also, using enumerations simplifies the process of reading from and writing to files using register mnemonics instead of address numbers as the spelling of the opcode can be extracted from the enumeration. For this reason instruction op-codes are also implemented as an enumeration.

Primitive routines are not implemented in the `Machine` class as all routines are handled by the `Interpreter` class. The addresses of the primitive routines however are implemented in the `Machine` class as an enumeration `PrimitiveRoutines`. This is done for consistency and code readability.

As Listing 3.26 on the following page shows, the `Instruction` class contains four variables. The variable `d` is implemented as an `Object` so that it may contain both integer and floating-point values.

The `read` and `write` methods are used respectively for reading and writing instructions to and from files.

The `Interpreter` class executes all of the instructions and primitive routines in the machine's code store. Due to the number of primitive routines and instructions, which are implemented similarly, all of the implementations will not be shown to avoid repetition.

Listing 3.27 on the next page shows the main loop of the interpreter which is placed in a method called `executeProgram`. This method is called immediately after the interpreter has loaded the machine's code store with instructions from an external file. The loop will continue running while the global `status` variable (implemented as an enumeration) is set to `Status.Running`. The status will change if the `HALT` instruction is executed or if an error oc-

```
1  public class Instruction {
2     public Machine.Opcode op;
3     public int n;
4     public Object d;
5     public Machine.RegisterAddress r;
6     .
7     .
8     .
9     public String write() {
10       ...
11    }
12
13    public static Instruction read(String codeLine) {
14       ...
15    }
16 }
```

Listing 3.26: The `Instruction` class.

```
1  static void executeProgram() {
2     Instruction instruction;
3     do {
4        instruction = codeStore[Machine.CP];
5
6        switch(instruction.op) {
7           case LOAD:
8              ...
9           case LOADL:
10             ...
11          .
12          .
13          .
14          case HALT:
15             status = Status.Halted;
16             break;
17       }
18    } while(status.equals(Status.Running));
19 }
```

Listing 3.27: The `executeProgram` method.

curs. Inside the loop there is a switch on the current instruction which checks the instruction's op-code and executes the correct block of code for that op-code.

Instructions

The implementation of the `LOAD` instruction is shown in Listing 3.28 on the facing page. First the operand address is calculated from the *d* and *r* fields of the instruction. The `getContent` method is an auxiliary method used for retrieving the contents of registers corresponding to the register addresses in the instructions. The `checkSpace` method is another auxiliary method used for checking whether there is space in the data store for the requested `LOAD` operation. If there is insufficient space on the stack the method changes the `status` variable to `Status.FailedDataStoreFull`, which halts the program after the execution of the current loop. Next *n* objects from the operand address are pushed on to the stack top, the ST register is incremented by n, and the CP register is set to point to the next instruction.

```
case LOAD:
   opAddress = instruction.d() + getContent(instruction.r);

   checkSpace(instruction.n);

   if(status.equals(Status.Running)) {
      for(int i = 0; i < instruction.n; i++) {
         dataStore[Machine.ST + i] = dataStore[opAddress + i];
      }

      Machine.ST += instruction.n;
      Machine.CP++;
   }

   break;
```

Listing 3.28: The LOAD instruction.

The STORE instruction shown in Listing 3.29 is implemented similarly to the LOAD instruction except it works in an opposite fashion. The first step is calculating the operand address, which is done in the same way as in the LOAD instruction. The ST register is decremented by n. This simulates popping n values from the stack. Then the top n values are stored at the operand address. Lastly the CP register is set to point to the next instruction.

```
case STORE:
   opAddress = instruction.d() + getContent(instruction.r);
   Machine.ST -= instruction.n;

   for(int i = 0; i < instruction.n; i++) {
      dataStore[opAddress + i] = dataStore[Machine.ST + i];
   }

   Machine.CP++;
   break;
```

Listing 3.29: The STORE instruction.

Listing 3.30 shows how the PUSH instruction is implemented. First the remaining space on the stack is checked to make sure it can accommodate the space requirements of the PUSH instruction. Next the ST register is incremented by d which is the number of objects to be pushed on to the stack. Lastly the CP register is incremented to point to the next instruction.

```
case PUSH:
   checkSpace(instruction.d());
   Machine.ST += instruction.d();
   Machine.CP++;
   break;
```

Listing 3.30: The PUSH instruction.

The POP instruction shown in Listing 3.31 on the next page first calculates the operand address then subtracts n from it. It then pops the n top values of the stack as the result and stores

them at the operand address. The ST register is then set to the operand address + n to simulate popping all the values above the result. The CP register is then incremented to point to the next instruction.

```
1  case POP:
2     opAddress = Machine.ST - instruction.n - instruction.d();
3     Machine.ST -= instruction.n;
4
5     for(int i = 0; i < instruction.n; i++) {
6        dataStore[opAddress + i] = dataStore[Machine.ST + i];
7     }
8
9     Machine.ST = opAddress + instruction.n;
10    Machine.CP++;
11    break;
```

Listing 3.31: The POP instruction.

Listing 3.32 shows the JUMP instruction which makes the CP register point to the operand address. The JUMPIF instruction is implemented in a similar fashion except it checks whether d and n are equal first. If they are, it does the same as JUMP and if they are not, it simply increments the CP register to point to the next instruction.

```
1  case JUMP:
2     Machine.CP = instruction.d() + getContent(instruction.r);
3     break;
```

Listing 3.32: The JUMP instruction.

The CALL instruction shown in Listing 3.33 on the next page checks whether the operand address points to a primitive routine or a code routine. If it points to a primitive routine it calls the corresponding routine using the `callPrimitiveRoutine` method, and increments the CP register to point to the next instruction. If the operand address points to a code routine it checks if there is room to push the routine's link data on to the stack. The dynamic link is then set to the current address held in the LB register, and the return address is set to be the next instruction. The LB register is then set to point to the current stack top, while the stack top is incremented by the link data size. The CP register is then set to point to the operand address which is the bottom of the routine body in the code store.

The implementation of the RETURN instruction shown in Listing 3.34 on the facing page is similar to that of the CALL instruction. The CP register is set to the return address and the LB register is set to the address in the dynamic link, both which have been set previously by a CALL instruction. The result is then popped and stored where the routine's arguments start. Then the rest of the values above the result are popped, which means all the routine's arguments and any other values left by the routine.

```
1  case CALL:
2     opAddress = instruction.d() + getContent(instruction.r);
3
4     if(opAddress >= Machine.PB) {
5        callPrimitiveRoutine(opAddress - Machine.PB);
6        Machine.CP++;
7     } else {
8        checkSpace(Machine.linkDataSize);
9
10       if(status.equals(Status.Running)) {
11          dataStore[Machine.ST] = Machine.LB;
12          dataStore[Machine.ST + 1] = Machine.CP + 1;
13          Machine.LB = Machine.ST;
14          Machine.ST = Machine.ST + linkDataSize;
15          Machine.CP = opAddress;
16       }
17    }
18
19    break;
```

Listing 3.33: The CALL instruction.

```
1  case RETURN:
2     opAddress = Machine.LB - instruction.d();
3     Machine.CP = (Integer)dataStore[Machine.LB + 1];
4     Machine.LB = (Integer)dataStore[Machine.LB];
5     Machine.ST = Machine.ST - instruction.n;
6
7     for (int i = 0; i < instruction.n; i++) {
8        dataStore[opAddress + i] = dataStore[Machine.ST + i];
9     }
10
11    Machine.ST = opAddress + instruction.n;
12    break;
```

Listing 3.34: The RETURN instruction.

Primitive Routines

The primitive routines in the `Interpreter` class are called with the `callPrimitiveRoutine` method. The method shown in Listing 3.35 on the next page takes a primitive routine address as input and then executes the corresponding primitive routine. The selection of primitive routines is done with a switch on the primitive routine address which executes the primitives routines corresponding block of code.

The `ADD` primitive routine shown in Listing 3.36 on the following page adds the two top values of the stack and replaces them with the result. Before any arithmetic operation, type-checking must be performed. If the first value is an integer both values are treated as integers, otherwise both values are treated as floating points. The implementations for `SUB`, `MULT`, `DIV` and `MOD` are similar. `DIV` and `MOD` however require an extra check to make sure division by zero does not occur. The methods `toInteger` and `toFloat` are auxiliary methods which are implemented for easier type-casting and exception handling.

Listing 3.37 on the next page shows the implementation of the `AND` primitive routine. It checks

```
1  static void callPrimitiveRoutine(int address) {
2     PrimitiveRoutine pRoutine = PrimitiveRoutine.get(address);
3
4     switch(pRoutine) {
5        case ADD:
6           ...
7        case SUB:
8           ...
9        .
10       .
11       .
12       // Rest of primitive routines.
13    }
14 }
```

Listing 3.35: The `callPrimitiveRoutine` method.

```
1  case ADD:
2     Machine.ST -= Machine.valueSize;
3
4     if(dataStore[Machine.ST - Machine.valueSize] instanceof Integer) {
5        dataStore[Machine.ST - Machine.valueSize] = toInteger(dataStore[Machine.ST -
              Machine.valueSize]) + toInteger(dataStore[Machine.ST]);
6     } else {
7        dataStore[Machine.ST - Machine.valueSize] = toFloat(dataStore[Machine.ST -
              Machine.valueSize])  + toFloat(dataStore[Machine.ST]);
8     }
9
10    break;
```

Listing 3.36: The ADD primitive routine.

whether the top two values on the stack are both true and replaces them both with the result. The methods `trueCheck` and `boolToInt` are auxiliary methods. `trueCheck` takes an integer as argument and returns a boolean value. It returns true if its argument matches the machine's integer representation of the true value, and false if it does not. `boolToInt` converts a `Boolean` object into an integer corresponding to the machine's true and false representations. The implementations of `LT`, `LE`, `GT` and `GE` work similarly to the `AND` primitive routine but use the `boolToInt` method to return their boolean results.

```
1  case AND:
2     Machine.ST -= Machine.valueSize;
3     dataStore[Machine.ST - Machine.valueSize] = boolToInt(trueCheck(dataStore[Machine.ST -
           Machine.valueSize]) && trueCheck(dataStore[Machine.ST]));
4     break;
```

Listing 3.37: The AND primitive routine.

The `EQ` routine shown in Listing 3.38 on the facing page checks whether the two top values of the stack are equal and replaces them with the boolean result. The `NEQ` is implemented the same way but returns the opposite boolean result.

```
case EQ:
   Machine.ST -= Machine.valueSize;
   dataStore[Machine.ST - Machine.valueSize] = boolToInt(dataStore[Machine.ST -
        Machine.valueSize].equals(dataStore[Machine.ST]));
   break;
```

Listing 3.38: The EQ primitive routine.

The WRITEINT primitive routine removes the top value from the stack and prints it on the screen. Listing 3.39 shows the implementation. The primitive routines WRITEFLOAT, WRITECHAR and WRITEBOOL are implemented similarly.

```
case WRITEINT:
   Machine.ST -= Machine.valueSize;
   System.out.print(dataStore[Machine.ST]);
```

Listing 3.39: The WRITEINT primitive routine.

The READINT primitive routine reads an integer input and pushes it onto the stack. If the input is not in the correct format it sets the `status` variable to `Status.FailedIOError` which halts the program on the next loop. Listing 3.40 shows the implementation. The primitive routines READFLOAT, READCHAR and READBOOL are implemented similarly.

```
case READINT:
   try {
      dataStore[Machine.ST] = scanner.nextInt();
   } catch(Exception e) {
      status = Status.FailedIOError;
   }

   Machine.ST += Machine.valueSize;
   break;
```

Listing 3.40: The READINT primitive routine.

Listing 3.41 shows the TOINT primitive routine which replaces the top value on the stack with a corresponding integer. The primitive routines TOFLOAT and TOCHAR work in a similar way.

```
case TOINT:
   dataStore[Machine.ST - Machine.valueSize] = toInteger(dataStore[Machine.ST -
        Machine.valueSize]);
```

Listing 3.41: The TOINT primitive routine.

The SIZEOF primitive routine is used for retrieving the size of arrays. It expects the address of an array to be on the top of the stack when it is called and replaces the address with the value at that address. This works because the size of an array is always placed at the array's address.

The implementation of the primitive routine is shown in Listing 3.42.

```
1 case SIZEOF:
2    dataStore[Machine.ST - Machine.valueSize] = dataStore[toInteger(dataStore[Machine.ST -
         Machine.valueSize])];
3    break;
```

Listing 3.42: The SIZEOF primitive routine.

The CHECKINDEXBOUNDS primitive routine is used for checking whether a requested index of an array is actually within the bounds of the array. Therefore it expects two values to be on the top of the stack when it is called: The requested index and the size of the array. It checks if the requested index is greater than zero and less than the size of the array. If it is, it removes the size of the array from the stack. If not, it changes the `status` variable to `Status.FailedIndexOutOfBounds`, which halts the program on the next loop. The implementation of the primitive routine is shown in Listing 3.43.

```
1 case CHECKINDEXBOUNDS:
2    Machine.ST -= Machine.valueSize;
3    if((toInteger(dataStore[Machine.ST - Machine.valueSize]) < 0) ||
         (toInteger(dataStore[Machine.ST - Machine.valueSize]) >=
         toInteger(dataStore[Machine.ST]))) {
4       status = Status.FailedIndexOutOfBounds;
5    }
6    break;
```

Listing 3.43: The CHECKINDEXBOUNDS primitive routine.

3.6 Code Generation

Code generation is the last phase of the KAL compiler. The purpose of the code generator is to traverse the AST and generate a KIL program that is semantically equivalent to the input KAL source code. After a successful run the code generator will output a file containing the KIL program, which can be run by the KVM.

When developing the code generator we used *code templates* outlining how different constructs of KAL should be translated to KIL code. Those relevant are included throughout this section.

The code generator consists of the class `CodeGenerator`, containing a number of auxiliary methods along with all the methods defined in the `Visitor` interface (see Section 3.2.4 on page 26), which are used to traverse the AST and generate KIL code. Furthermore, the code generator consists of a number of classes containing information such as addresses and values of various runtime entities. These are:

- `KnownAddress` - Used for variables, arrays and formal parameters.
- `KnownValue` - Used for constants.

- `KnownRoutine` - Used for functions and procedures.
- `PrimitiveRoutine` - Used for primitive routines.

When the code generator is initialized its constructor shown in Listing 3.44 will be called. The constructor calls the method `addStdEnvironment`, which will initialize the standard environment.

```
1 public CodeGenerator() {
2    addStdEnvironment();
3 }
```

Listing 3.44: The `CodeGenerator` class constructor.

The method `addStdEnvironment` shown in Listing 3.45 creates new runtime entities containing either the address of a primitive routine or the value of a constant.

```
1  private final void  initializeStdEnvironment() {
2     StdEnvironment.trueDecl.entity = new KnownValue(Machine.trueRep);
3     StdEnvironment.falseDecl.entity = new KnownValue(Machine.falseRep);
4     StdEnvironment.addDecl.entity = new PrimitiveRoutine(Machine.PrimitiveRoutine.ADD);
5     StdEnvironment.subtractDecl.entity = new
           PrimitiveRoutine(Machine.PrimitiveRoutine.SUB);
6     .
7     .
8     .
9     // The rest of the primitive routines.
10 }
```

Listing 3.45: The `initializeStdEnvironment` method.

To start the code generation process the method `generateCode` is called, which is shown in Listing 3.46 on the next page. First the `visit` method of the Program AST is called, which will start the traversal of the entire AST translating it into a KIL program. By this time a procedure with the name "Run" will have been visited and its location stored in the variable `runDisplacement`. The presence of this procedure is ensured by the contextual analyzer. The "Run"-procedure is the equivalent in KAL of the "main"-method in C, and will implicitly be called first when a program is executed. After this, all globally allocated stack space is deallocated and all applied occurrences of forward declared functions and procedures are patched with the correct addresses in a fashion similar to the "Run"-procedure. Finally, a `HALT` instruction is issued to stop the interpreter.

Listing 3.47 on the following page shows the method `addInstruction`. This method is used to add instructions to the KIL program and is used throughout the code generator. The method is overloaded to provide support for floating-point numbers as well as integers.

```
1  public void generateCode(Program progAST) {
2     progAST.visit(this, new Frame(0, 0));
3     addInstruction(Opcode.CALL, RegisterAddress.SB.ordinal(), runDisplacement,
          RegisterAddress.CB);
4
5     if(globalExtraSize > 0) {
6        addInstruction(Opcode.POP, 0, globalExtraSize, null);
7     }
8
9     patchForwardDeclarations();
10    addInstruction(Opcode.HALT, 0, 0, null);
11 }
```

Listing 3.46: The `generateCode` method.

```
1  private void addInstruction(Opcode op, int n, int d, RegisterAddress r) {
2     Instruction nextInstruction = new Instruction();
3
4     nextInstruction.op = op;
5     nextInstruction.n = n;
6     nextInstruction.d = d;
7     nextInstruction.r = r;
8
9     Machine.codeStore[nextInstructionAddress] = nextInstruction;
10    nextInstructionAddress++;
11 }
```

Listing 3.47: The `addInstruction` method.

3.6.1 Declarations

In KAL there are five different kinds of declarations: Variable, array, constant, function and procedure declarations. Constant, function and procedure declarations can only happen globally.

Variable Declarations

Listing 3.48 on the next page shows the code template for variable declarations and Listing 3.49 on the facing page shows how they are implemented. First the size of the variable is determined in order to expand the stack. This is done by visiting the type denoter of the variable. After this, a new `KnownAddress` entity is created containing the scope and address of the variable. The entity will be used later in all applied occurrences of the variable. Each variable in KAL is declared with a default value according to the type of the variable. Default values for different types are shown in Table 3.6 on the next page.

The auxiliary method `storeDefaultValue` stores the corresponding default value at the variable address. Finally, the size of the variable is added to the variable `globalExtraSize`, which will be used to deallocate space from the stack just before the program finishes running.

Type	Value
`int`	0
`float`	0.0
`char`	ASCII character 0
`bool`	`false`

Table 3.6: Default values for variables in KAL.

```
execute ⟨ TypeDenoter Identifier ⟩ =
      PUSH s                                                    where s = size of TypeDenoter
```

Listing 3.48: Code template for variable declarations.

```
1  public Object visitVariableDeclaration(VariableDeclaration ast, Object arg) {
2     Frame frame = (Frame) arg;
3
4     int expandSize = (Integer) ast.tAST.visit(this, null);
5     addInstruction(Opcode.PUSH, 0, expandSize, null);
6     ast.entity = new KnownAddress(Machine.addressSize, frame.level, globalExtraSize);
7
8     storeDefaultValue(ast, frame, expandSize);
9     globalExtraSize += expandSize;
10
11    return null;
12 }
```

Listing 3.49: The `visitVariableDeclaration` method.

Function Declarations

Listing 3.50 shows the code template for function declarations and Listing 3.51 on the next page shows how they are implemented. Procedure declarations are almost identical, except they do not have a return value. The code generator starts by adding an empty jump instruction (the correct address will be inserted at the end), which will skip the the entire body of the function to make sure that the function is not executed when it is declared but instead when it is called. The size and address of the function is stored in a new `KnownRoutine` entity. The formal parameters are visited and represented as `KnownAddress` entities and can be accessed with a negative displacement relative to their local base. Finally, the commands of the function are visited.

```
execute ⟨ function TypeDenoter Identifier ( FormalParameterSequence ) { Command } ⟩ =
      JUMP g
      execute Command
   g:
```

Listing 3.50: Code template for function declarations.

```
1  public Object visitFunctionDeclaration(FunctionDeclaration ast, Object arg) {
2     if(ast.cAST != null) {
3        Frame frame = (Frame) arg;
4
5        int jumpAddress = nextInstructionAddress;
6        int extraSize;
7        parameterDeclSize = 0;
8        returnDeclSize = 0;
9        addInstruction(Opcode.JUMP, 0, 0, RegisterAddress.CB);
10       returnDeclSize = (Integer)ast.tdAST.visit(this, null);
11       routineDeclarations.put(ast.iAST.spelling, nextInstructionAddress);
12       ast.entity = new KnownRoutine(Machine.closureSize, nextInstructionAddress);
13
14       Frame formalParameterFrame = new Frame(frame.level + 1, 0);
15
16       if(ast.fpsAST != null) {
17          parameterDeclSize = (Integer)ast.fpsAST.visit(this, formalParameterFrame);
18       }
19
20       Frame commandFrame = new Frame(frame.level + 1, Machine.linkDataSize -
              Machine.addressSize);
21
22       for(Command command : ast.cAST.cASTList) {
23          extraSize = (Integer) command.visit(this, commandFrame);
24          commandFrame = new Frame(commandFrame.level, commandFrame.size + extraSize);
25       }
26
27       patch(jumpAddress, nextInstructionAddress);
28    }
29
30    return null;
31 }
```

Listing 3.51: The `visitFunctionDeclaration` method.

3.6.2 Commands

Commands such as assignments, control structures and calls to functions or procedures appear in the bodies of functions and procedures.

Assignments

Listing 3.52 shows the code template for assignments and Listing 3.53 on the next page shows how they are implemented. When visiting an assignment the code generator will start by determining the size of and evaluating the expression on the right-hand side so that the result is pushed on to the stack immediately before calling the `store` method, which is shown in Listing 3.54 on the facing page with some details omitted. The `store` method will add a `STORE` instruction, which will pop the result (size is known) of the expression off the stack and store it at the address of the left-hand side identifier.

```
execute ⟨ Identifier = Expression ⟩ =
      evaluate Expression
      STORE a                                          where a = address of Identifier
```

Listing 3.52: Code template for assignments.

```
1  public Object visitAssignCommand(AssignCommand ast, Object arg) {
2     Frame frame = (Frame) arg;
3
4     int valueSize = (Integer) ast.eAST.visit(this, frame);
5     store(ast.iAST, frame, valueSize);
6
7     return 0;
8  }
```

Listing 3.53: The `visitAssignCommand` method.

```
1  private void store(Identifier ast, Frame frame, int valueSize) {
2     RuntimeEntity baseObject = (RuntimeEntity)ast.visit(this, frame);
3
4     if(baseObject instanceof KnownAddress) {
5        ObjectAddress address = ((KnownAddress) baseObject).address;
6        .
7        .
8        .
9        addInstruction(Opcode.STORE, valueSize, address.displacement,
              RegisterAddress.get(displayRegister(frame.level, address.level)));
10    }
11 }
```

Listing 3.54: The `store` method.

If-Else Statements

Listing 3.55 shows the code template for if-else statements. If-else statements follow a simple pattern, which is: First evaluate the expression and push the result on to the stack. If the expression evaluates to true (1), the first command is executed and the second command is skipped. If the expression evaluates to false (0), the first command is skipped and the second command is executed.

```
execute ⟨ if ( Expression ) { Command1 } else { Command2 } ⟩ =
      evaluate Expression
      JUMPIF(false) g
      execute Command1
      POP s1                        where s1 = size of allocated stack space within Command1
      JUMP h
   g: execute Command2
      POP s2                        where s2 = size of allocated stack space within Command2
   h:
```

Listing 3.55: Code template for if-else statements.

Listing 3.56 on the following page shows how if-else statements are implemented. The implementation follows the code template very closely. One thing to note is that the `POP` instruction only works because all commands which allocate stack space will return the amount of stack space allocated, which is added cumulative to the variable `extraSize`. This variable determines the amount of stack space to be deallocated.

```
1  public Object visitIfElseCommand(IfElseCommand ast, Object arg) {
2     Frame frame = (Frame) arg;
3     int jumpAddress = 0;
4     int jumpIfAddress = 0;
5     int extraSize = 0;
6
7     ast.eAST.visit(this, frame);
8     jumpIfAddress = nextInstructionAddress;
9     addInstruction(Opcode.JUMPIF, Machine.falseRep, 0, RegisterAddress.CB);
10
11    for(Command command : ast.c1AST.cASTList) {
12       extraSize += (Integer) command.visit(this, frame);
13    }
14
15    jumpAddress = nextInstructionAddress;
16    addInstruction(Opcode.JUMP, 0, 0, RegisterAddress.CB);
17
18    if(extraSize > 0) {
19       addInstruction(Opcode.POP, 0, extraSize, null);
20    }
21
22    extraSize = 0;
23    patch(jumpIfAddress, nextInstructionAddress);
24
25    for(Command command : ast.c2AST.cASTList) {
26       extraSize += (Integer) command.visit(this, frame);
27    }
28
29    if(extraSize > 0) {
30       addInstruction(Opcode.POP, 0, extraSize, null);
31    }
32
33    patch(jumpAddress, nextInstructionAddress);
34
35    return 0;
36 }
```

Listing 3.56: The `visitIfElseCommand` method.

Function calls

Listing 3.57 shows the code template for function calls. When a function is called all of its parameters are evaluated and pushed on to the stack and execution is shifted to the beginning of the function declaration.

```
execute ⟨ Identifier ( ActualParameterSequence ) ⟩ =
      evaluate ActualParameterSequence
      CALL(SB) a                                  where a = address of function entry point
```

Listing 3.57: Code template for function calls.

Listing 3.57 shows how function calls are implemented. First, the actual parameter sequence is visited, which will evaluate all parameters (expressions) and push the results on to the stack. To issue a `CALL` instruction to the correct address the identifier is visited, which has a `KnownRoutine` entity bound to it containing the address of the function.

```
1  public Object visitCallCommand(CallCommand ast, Object arg) {
2     Frame frame = (Frame) arg;
3     int parameterSize = 0;
4
5     if(ast.apsAST != null) {
6        parameterSize = (Integer) ast.apsAST.visit(this, frame);
7     }
8
9     ast.iAST.visit(this, new Frame(frame.level, parameterSize));
10
11    return 0;
12 }
```

Listing 3.58: The `visitCallCommand` method.

3.6.3 Expressions

In KAL there are 7 different kinds of expressions: `IntegerExpression`, `FloatExpression`, `CharacterExpression`, `BooleanExpression`, `VariableExpression`, `CallExpression` and `ArrayExpression`. For example, "5" is an `IntegerExpression` and "x" is a `VariableExpression`. When any of these expressions are used in conjunction with a binary operator such as "+" or "-" they are encapsulated in a `BinaryExpression`, and when used with the unary operator "!" (logical not) they are encapsulated in a `UnaryExpression`. Binary expressions may contain unary expressions and vice versa.

In KAL expressions are evaluated from left to right without any operator precedence. If operator precedence is desired parentheses must be used.

Binary Expressions

Listing 3.59 on the next page shows the code template for binary expressions. A binary expression consists of two expressions and an operator. As mentioned earlier, these expressions may in turn be binary expressions and so forth. To evaluate a simple expression such as "1 + 2", the integer literals 1 and 2 will be pushed on to the stack and the primitive routine `ADD` will be called. A more complex expression such as "1 + (2 * 3)" will be represented in the AST as shown in Figure 3.10 on the following page.

Following the code template for binary expressions, we see that when the first expression is evaluated the integer literal 1 will be pushed on to the stack. After this the second expression is evaluated, which is a binary expression, resulting in the integer literals 2 and 3 being pushed on to the stack. The operator in the inner binary expression is then called, multiplying 2 and 3 and pushing the result on to the stack, after which the operator from the outer binary expression is called, adding 1 to the topmost value of the stack, which is the result of multiplying 2 and 3. The KIL code used to evaluate the expression "1 + (2 * 3)" is shown in Listing 3.60 on the next page.

The implementation of binary expressions is shown in Listing 3.61 on the following page and follows the pattern described above. The first expression is evaluated and afterwards the second expression is evaluated. At the end the operator is called. In Listing 3.45 on page 55 we

```
evaluate ⟨ Expression1 Operator Expression2 ⟩ =
      evaluate Expression1
      evaluate Expression2
      CALL p                                        where p = primitive routine (operator)
```

Listing 3.59: Code template for binary expressions.

```
LOADL 0  1  0     \\ Load 1.
LOADL 0  2  0     \\ Load 2.
LOADL 0  3  0     \\ Load 3.
CALL  4  3  PB    \\ Multiply 2 and 3.
CALL  4  1  PB    \\ Add 1 and 6.
```

Listing 3.60: KIL code used to evaluate the expression "1 + (2 * 3)".

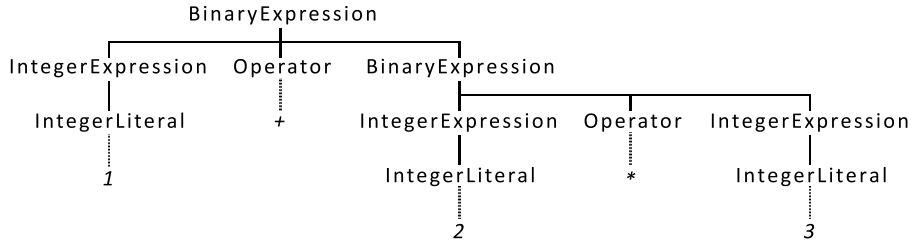

Figure 3.10: AST representing the expression "1 + (2 * 3)".

showed how each operator was represented as `PrimitiveRoutine` entity. When visiting an operator, the code generator fetches its entity and calls the appropriate primitive routine. Unary expressions are handled in the same way, except they only contain one expression.

```
1  public Object visitBinaryExpression(BinaryExpression ast, Object arg) {
2     Frame frame = (Frame) arg;
3
4     int valueSize = (Integer) ast.type.visit(this, null);
5     int valueE1Size = (Integer) ast.e1AST.visit(this, frame);
6     Frame frameE1 = new Frame(frame.level, valueE1Size);
7
8     int valueE2Size = (Integer) ast.e2AST.visit(this, frameE1);
9     Frame frameE2 = new Frame(frame.level, valueE2Size);
10
11    ast.oAST.visit(this, frameE2);
12
13    return valueSize;
14 }
```

Listing 3.61: The `visitBinaryExpression` method.

Variable Expressions

Listing 3.62 shows the code template for variable expressions. Variable expressions can appear by themselves in expressions such as "x" or as part of a binary expression in expressions such as "x + y". Variable expressions are handled by loading the value of the variable and pushing it on to the stack.

```
evaluate ⟨ Identifier ⟩ =
      LOAD a                                          where a = address of Identifier
```

Listing 3.62: Code template for variable expressions.

Listing 3.63 shows how variable expressions are implemented. The code generator handles variable expressions by first finding the size of the variable and then calling the `fetch` method with the identifier, frame and size of the variable. The relevant parts of the `fetch` method are shown in Listing 3.64. The value of the variable will be retrieved by getting the displacement of the variable relative to the current scope from the `KnownAddress` entity bound to the identifier of the variable and adding a `LOAD` instruction with this displacement as well as the size and scope.

```
1  public Object visitVariableExpression(VariableExpression ast, Object arg) {
2     Frame frame = (Frame) arg;
3
4     int valueSize = (Integer) ast.type.visit(this, null);
5     fetch(ast.iAST, frame, valueSize);
6
7     return valueSize;
8  }
```

Listing 3.63: The `visitVariableExpression` method.

```
1  private void fetch(Identifier ast, Frame frame, int valueSize) {
2     RuntimeEntity baseObject = (RuntimeEntity) ast.visit(this, frame);
3
4     if(baseObject instanceof KnownAddress) {
5        ObjectAddress address = ((KnownAddress) baseObject).address;
6        if(ast.decl instanceof VariableDeclaration) {
7           addInstruction(Opcode.LOAD, valueSize, address.displacement,
                 RegisterAddress.get(displayRegister(address.level)));
8        }
9        .
10       .
11       .
12       // Code for fetching array values.
13    }
14    .
15    .
16    .
17 }
```

Listing 3.64: The `fetch` method.

CHAPTER 4

CONCLUSION

The purpose of this report was to document the design of a programming language and the implementation of a compiler and a virtual machine to run the compiled code on.

4.1 Language Specification

The syntax of KAL was specified formally using BNF/EBNF notation, and the semantics using a combination of informal and structural operational semantics.

4.1.1 Evaluation Criteria

The evaluation criteria for KAL defined in Section 2.1 on page 4 stated that readability was of high importance, while writability and reliability was of medium importance.

We designed KAL to be as readable as possible by limiting the number of basic components, limiting feature multiplicity to the extent possible and by choosing a syntactic style that many people are already familiar with. We also included data types to increase readability.

To increase writability we included abstraction in the form of functions and procedures and to increase reliability we implemented static type checking and runtime exceptions.

4.1.2 Scope Rules

The scope rules for variables and routines in KAL are neither static nor dynamic in the traditional sense. Since routines and variables cannot be redeclared, it will never be necessary to differentiate between declarations. This approach eliminates confusion from the programmer's perspective regarding which variable and routine bindings are in effect. This will improve the readability of KAL, which has the highest priority in our evaluation criteria (see Section 2.1 on page 4). The downside is that the programmer will not be able to redeclare common variable names such as `i` and `j` for counters in while loops.

4.2 Implementation

The implementation of the KAL compiler and KVM was done in Java. The KAL compiler is a multi-pass compiler consisting of three parts, which compiles to the intermediate language KIL that can be executed in the virtual machine KVM.

4.2.1 Evaluation Criteria

The evaluation criteria for the KAL compiler and KVM were defined in Section 3.1 on page 20. In these we stated that correctness and portability were very important and flexibility and comprehensibility were important while performance was regarded as less important.

To achieve correctness we specified both the syntax the and semantics of KAL formally to avoid ambiguity. We also wrote small test programs in KAL for various scenarios to test if the compiled program behaved according to the semantics when run in the KVM.

At the time of writing, we are aware of one area of the KAL compiler, which does not comply with our semantics. The contextual analyzer does not report an error for expressions such as `true + true` or `true > false`. To fix this problem we could introduce a new type, `number`, in the contextual analyzer meaning either an integer or a floating-point value. Then, when the contextual analyzer encounters an arithmetic expression such as `true + true` it should ensure that both operands are of type `number`.

To make KAL programs as portable as possible we chose to interpret programs in the KVM, as opposed to compiling to the machine's assembler code. This enables KAL programs to be run on any machine on which the KVM has been implemented. Using an interpreter slowed down the execution of KAL programs. To minimize the performance impact we chose to compile KAL code to the intermediate language KIL, which allowed us to perform iterative interpretation.

To make the implementation of the KAL compiler as flexible as possible, we divided it into several independent modules, all relying only on the AST. This modular design also increased the comprehensibility.

4.2.2 Error Reporting

In KAL all compile-time error reports are generated by either the syntactic analyzer or the contextual analyzer. If an error is detected in the syntactic analyzer the entire compilation process will halt and an error report will be presented to the user. If an error is detected during contextual analysis it will be reported to the user, but the contextual analyzer will continue running until all errors are reported, whereafter the compilation process will halt. The reason why the syntactic analyzer will halt on the first error, is that in order to produce more error reports it would have to attempt to skip past the error and resume reading from a syntactically correct place in the code. This method would be a good alternative, but is prone to produce inaccurate error reports and may skip parts of the code if the error recovery algorithm is inefficient. The reason why the contextual analyzer will report all contextual errors before halting, is that it simply traverses the entire AST and reports errors as they are encountered.

4.2.3 Arrays

In KAL arrays are implemented statically, which means that the size of an array must be known at compile-time. This imposes some limitations on the operations which can be performed on arrays. One such limitation is that an array cannot be the return value of a function, since that would imply creating an array of arbitrary size. To overcome this limitation we have chosen to pass arrays by reference so that they can be manipulated within routines. Another solution would be to implement dynamic arrays, whose size is not known until run-time. This could be done by adding a heap to the KVM and storing the array in the heap as a linked list, where each element in the list points to the next element.

4.2.4 Instruction Format

At the moment KIL instructions are in the format “op n d r”, where “d” can be either an integer or a floating-point value. This makes it difficult for the interpreter to distinguish values from addresses as both are represented as integers. The interpreter relies on the code generator to place an address on the stack whenever an address is needed. By implementing a fifth field “type”, the interpreter would be able to differentiate between addresses and values and types of values. This would allow us to reduce the amount of redundant code used for type checking arithmetic operations in the primitive routines etc. and overall reduce the amount of redundant code in the interpreter.

APPENDIX A

SEMANTICS

A.1 Arithmetic Expressions

The transition rules for arithmetic expressions [**ArExp**] are shown in Table A.1.

[ADD]
$$\frac{env_E,\ sto \vdash a_1 \rightarrow_a v_1 \quad env_E,\ sto \vdash a_2 \rightarrow_a v_2}{env_E,\ sto \vdash a_1 \texttt{+} a_2 \rightarrow_a v}$$
where $v = v_1 + v_2$

[SUBTRACT]
$$\frac{env_E,\ sto \vdash a_1 \rightarrow_a v_1 \quad env_E,\ sto \vdash a_2 \rightarrow_a v_2}{env_E,\ sto \vdash a_1 \texttt{-} a_2 \rightarrow_a v}$$
where $v = v_1 - v_2$

[MULTIPLY]
$$\frac{env_E,\ sto \vdash a_1 \rightarrow_a v_1 \quad env_E,\ sto \vdash a_2 \rightarrow_a v_2}{env_E,\ sto \vdash a_1 \texttt{*} a_2 \rightarrow_a v}$$
where $v = v_1 \cdot v_2$

[DIVIDE]
$$\frac{env_E,\ sto \vdash a_1 \rightarrow_a v_1 \quad env_E,\ sto \vdash a_2 \rightarrow_a v_2}{env_E,\ sto \vdash a_1 \texttt{/} a_2 \rightarrow_a v}$$
where $v = \frac{v_1}{v_2}$

[MODULO]
$$\frac{env_E,\ sto \vdash a_1 \rightarrow_a v_1 \quad env_E,\ sto \vdash a_2 \rightarrow_a v_2}{env_E,\ sto \vdash a_1 \texttt{\%} a_2 \rightarrow_a v}$$
where $v = v_1 \bmod v_2$

Continued on the next page...

[PARENTHESIS]	$\dfrac{env_E,\ sto \vdash a_1 \rightarrow_a v_1}{env_E,\ sto \vdash (a_1) \rightarrow_a v_1}$
[NUMERAL]	$env_E,\ sto \vdash n \rightarrow_a v$ if $\mathcal{N}[\![n]\!] = v$ where $\mathcal{N} : \mathbf{Num} \rightarrow \mathbb{R}$
[VARIABLE]	$env_V,\ sto \vdash x \rightarrow_a v$ if $env_V\ x = l$ and $sto\ l = v$
[CONSTANT]	$env_C,\ sto \vdash c \rightarrow_a v$ if $env_C\ c = l$ and $sto\ l = v$
[ARRAY]	$env_A,\ sto \vdash r[a_1] \rightarrow_a v_2$ if $env_A\ r[v_1] = l$ and $sto\ l = v_2$ where $a_1 \rightarrow_a v_1$

Table A.1: Transition rules for arithmetic expressions [**ArExp**].

A.2 Boolean Expressions

The transition rules for boolean expressions [**BoolExp**] are shown in Table A.2.

[EQUALS-TRUE]	$\dfrac{env_E,\ sto \vdash e_1 \rightarrow_e v_1 \quad env_E,\ sto \vdash e_2 \rightarrow_e v_2}{env_E,\ sto \vdash e_1 == e_2 \rightarrow_b \text{true}}$ if $v_1 = v_2$
[EQUALS-FALSE]	$\dfrac{env_E,\ sto \vdash e_1 \rightarrow_e v_1 \quad env_E,\ sto \vdash e_2 \rightarrow_e v_2}{env_E,\ sto \vdash e_1 == e_2 \rightarrow_b \text{false}}$ if $v_1 \neq v_2$
[NOTEQUALS-TRUE]	$\dfrac{env_E,\ sto \vdash e_1 \rightarrow_e v_1 \quad env_E,\ sto \vdash e_2 \rightarrow_e v_2}{env_E,\ sto \vdash e_1\ != e_2 \rightarrow_b \text{true}}$

Continued on the next page...

if $v_1 \neq v_2$

[NOTEQUALS-FALSE]
$$\frac{env_E, sto \vdash e_1 \rightarrow_e v_1 \quad env_E, sto \vdash e_2 \rightarrow_e v_2}{env_E, sto \vdash e_1 \texttt{ != } e_2 \rightarrow_b \text{false}}$$

if $v_1 = v_2$

[GREATERTHAN-TRUE]
$$\frac{env_E, sto \vdash a_1 \rightarrow_a v_1 \quad env_E, sto \vdash a_2 \rightarrow_a v_2}{env_E, sto \vdash a_1 \texttt{ > } a_2 \rightarrow_b \text{true}}$$

if $v_1 > v_2$

[GREATERTHAN-FALSE]
$$\frac{env_E, sto \vdash a_1 \rightarrow_a v_1 \quad env_E, sto \vdash a_2 \rightarrow_a v_2}{env_E, sto \vdash a_1 \texttt{ > } a_2 \rightarrow_b \text{false}}$$

if $v_1 \leq v_2$

[GREATERTHANEQUALS-TRUE]
$$\frac{env_E, sto \vdash a_1 \rightarrow_a v_1 \quad env_E, sto \vdash a_2 \rightarrow_a v_2}{env_E, sto \vdash a_1 \texttt{ >= } a_2 \rightarrow_b \text{true}}$$

if $v_1 \geq v_2$

[GREATERTHANEQUALS-FALSE]
$$\frac{env_E, sto \vdash a_1 \rightarrow_a v_1 \quad env_E, sto \vdash a_2 \rightarrow_a v_2}{env_E, sto \vdash a_1 \texttt{ >= } a_2 \rightarrow_b \text{false}}$$

if $v_1 < v_2$

[LESSTHAN-TRUE]
$$\frac{env_E, sto \vdash a_1 \rightarrow_a v_1 \quad env_E, sto \vdash a_2 \rightarrow_a v_2}{env_E, sto \vdash a_1 \texttt{ < } a_2 \rightarrow_b \text{true}}$$

if $v_1 < v_2$

[LESSTHAN-FALSE]
$$\frac{env_E, sto \vdash a_1 \rightarrow_a v_1 \quad env_E, sto \vdash a_2 \rightarrow_a v_2}{env_E, sto \vdash a_1 \texttt{ < } a_2 \rightarrow_b \text{false}}$$

if $v_1 \geq v_2$

[LESSTHANEQUALS-TRUE]
$$\frac{env_E, sto \vdash a_1 \rightarrow_a v_1 \quad env_E, sto \vdash a_2 \rightarrow_a v_2}{env_E, sto \vdash a_1 \texttt{ <= } a_2 \rightarrow_b \text{true}}$$

if $v_1 \leq v_2$

Continued on the next page...

[LESSTHANEQUALS-FALSE]
$$\frac{env_E,\ sto \vdash a_1 \rightarrow_a v_1 \quad env_E,\ sto \vdash a_2 \rightarrow_a v_2}{env_E,\ sto \vdash a_1 \texttt{<=} a_2 \rightarrow_b \text{false}}$$
if $v_1 > v_2$

[NOT-TRUE]
$$\frac{env_E,\ sto \vdash b \rightarrow_b \text{true}}{env_E,\ sto \vdash \texttt{!}\, b \rightarrow_b \text{false}}$$

[NOT-FALSE]
$$\frac{env_E,\ sto \vdash b \rightarrow_b \text{false}}{env_E,\ sto \vdash \texttt{!}\, b \rightarrow_b \text{true}}$$

[AND-TRUE]
$$\frac{env_E,\ sto \vdash b_1 \rightarrow_b \text{true} \quad env_E,\ sto \vdash b_2 \rightarrow_b \text{true}}{env_E,\ sto \vdash b_1 \texttt{ \&\& } b_2 \rightarrow_b \text{true}}$$

[AND-FALSE]
$$\frac{env_E,\ sto \vdash b_1 \vee b_2 \rightarrow_b \text{false}}{env_E,\ sto \vdash b_1 \texttt{ \&\& } b_2 \rightarrow_b \text{false}}$$

[OR-TRUE]
$$\frac{env_E,\ sto \vdash b_1 \vee b_2 \rightarrow_b \text{true}}{env_E,\ sto \vdash b_1 \texttt{ || } b_2 \rightarrow_b \text{true}}$$

[OR-FALSE]
$$\frac{env_E,\ sto \vdash b_1 \wedge b_2 \rightarrow_b \text{false}}{env_E,\ sto \vdash b_1 \texttt{ || } b_2 \rightarrow_b \text{false}}$$

[PARENTHESIS]
$$\frac{env_E,\ sto \vdash b_1 \rightarrow_b v}{env_E,\ sto \vdash (b_1) \rightarrow_b v}$$

Table A.2: Transition rules for boolean expressions [**BoolExp**].

A.3 Commands

The transition rules for commands [**Com**] are shown in Table A.3.

[VAR-ASS] $env_E \vdash \left\langle x = e,\ sto \right\rangle \rightarrow sto\left[l \mapsto v\right]$

where $env_E,\ sto \vdash e \rightarrow_e v$
and $env_V\ x = l$

[ARRAY-ASS] $env_E \vdash \left\langle r[a] = e,\ sto \right\rangle \rightarrow sto\left[l \mapsto v_2\right]$

where $env_E,\ sto \vdash a \rightarrow_a v_1$
and $env_E,\ sto \vdash e \rightarrow_e v_2$
and $env_A\ r[v_1] = l$

Continued on the next page...

[IF-TRUE]
$$\frac{env_E \vdash \left\langle C,\ sto \right\rangle \to sto'}{env_E \left\langle \textbf{if}(b)\ \{C\},\ sto \right\rangle \to sto'}$$
if $env_E,\ sto \vdash b \to_b \text{true}$

[IF-FALSE]
$$env_E \vdash \left\langle \textbf{if}(b)\ \{C\},\ sto \right\rangle \to sto$$
if $env_E,\ sto \vdash b \to_b \text{false}$

[IF-ELSE-TRUE]
$$\frac{env_E \vdash \left\langle C_1,\ sto \right\rangle \to sto'}{env_E \vdash \left\langle \textbf{if}(b)\ \{C_1\}\ \textbf{else}\ \{C_2\},\ sto \right\rangle \to sto'}$$
if $env_E,\ sto \vdash b \to_b \text{true}$

[IF-ELSE-FALSE]
$$\frac{env_E \vdash \left\langle C_2,\ sto \right\rangle \to sto'}{env_E \vdash \left\langle \textbf{if}(b)\ \{C_1\}\ \textbf{else}\ \{C_2\},\ sto \right\rangle \to sto'}$$
if $env_E,\ sto \vdash b \to_b \text{false}$

[WHILE-TRUE]
$$\frac{env_E \vdash \left\langle C,\ sto \right\rangle \to sto'' \quad env_E \vdash \left\langle \textbf{while}(b)\ \{C\},\ sto'' \right\rangle \to sto'}{env_E \vdash \left\langle \textbf{while}(b)\ \{C\},\ sto \right\rangle \to sto'}$$
if $env_E,\ sto \vdash b \to_b \text{true}$

[WHILE-FALSE]
$$env_E \vdash \left\langle \textbf{while}(b)\ \{C\},\ sto \right\rangle \to sto$$
if $env_E,\ sto \vdash b \to_b \text{false}$

[CALL-R-PROC]
$$\frac{env'_V,\ env'_C,\ env'_A\left[FPS \mapsto l^*\right]\left[\text{next} \mapsto l'\right],\ env'_F,\ env'_P \vdash \left\langle C,\ sto \right\rangle \to sto'}{env_E,\ env_P \vdash \left\langle p(APS),\ sto \right\rangle \to sto'}$$
where $env_P\ p = (C,\ FPS,\ env'_E,\ env'_P)$
and $l^* = env_A\ APS$
and $l' = env_A\ \text{next}$

Continued on the next page...

[CALL-R-FUNC]
$$\frac{env'_V,\ env'_C,\ env'_A\Big[FPS \mapsto l^*\Big]\Big[\text{next} \mapsto l'\Big],\ env'_P,\ env'_F \vdash \Big\langle C,\ sto\Big\rangle \to sto'}{env_P,\ env_E \vdash \Big\langle f(APS),\ sto\Big\rangle \to sto'}$$

where $env_F\ f = (C,\ T,\ FPS,\ env'_E,\ env'_P)$
and $l^* = env_A\ APS$
and $l' = env_A$ next

[CALL-V-PROC]
$$\frac{env''_V,\ env'_C,\ env'_A,\ env'_F,\ env'_P \vdash \Big\langle C,\ sto\Big[l^* \mapsto v^*\Big]\Big\rangle \to sto'}{env_V,\ env_C,\ env_F,\ env_P \vdash \Big\langle p(APS),\ sto\Big\rangle \to sto'}$$

where $env_P\ p = (C,\ FPS,\ env'_E,\ env'_P)$
and $env_V,\ env_C,\ env_F,\ sto \vdash APS \to_{APS} v^*$
and $l = env_V$ next
and $env''_V = env'_V\Big[FPS \mapsto l^*\Big]\Big[\text{next} \mapsto \text{new}\ l\Big]$

[CALL-V-FUNC]
$$\frac{env''_V,\ env'_C,\ env'_A,\ env'_P,\ env'_F \vdash \Big\langle C,\ sto\Big[l^* \mapsto v^*\Big]\Big\rangle \to sto'}{env_V,\ env_C,\ env_F \vdash \Big\langle f(APS),\ sto\Big\rangle \to sto'}$$

where $env_F\ f = (C,\ T,\ FPS,\ env'_E,\ env'_P)$
and $env_V,\ env_C,\ env_F,\ sto \vdash APS \to_{APS} v^*$
and $l = env_V$ next
and $env''_V = env'_V\Big[FPS \mapsto l^*\Big]\Big[\text{next} \mapsto \text{new}\ l\Big]$

Table A.3: Transition rules for commands [**Com**].

A.4 Declarations

The transition rules for declarations are shown in Table A.4.

[VAR-DECL]
$$\frac{\Big\langle VD,\ env_V\Big[x \mapsto l\Big]\Big[\text{next} \mapsto \text{new}\ l\Big],\ sto\Big\rangle \to_{VD} (env'_V,\ sto)}{\Big\langle T\ x,\ env_V,\ sto\Big\rangle \to_{VD} (env'_V,\ sto)}$$

where $l = env_V$ next

[CONST-DECL]
$$\frac{\Big\langle CD,\ env_C\Big[c \mapsto l\Big]\Big[\text{next} \mapsto \text{new}\ l\Big],\ sto\Big[l \mapsto v\Big]\Big\rangle \to_{CD} (env'_C,\ sto')}{\Big\langle \textbf{constant}\ T\ c = e,\ env_C,\ sto\Big\rangle \to_{CD} (env'_C,\ sto')}$$

Continued on the next page...

where $env_C, sto \vdash e \rightarrow_e v$
and $l = env_C$ next

[ARRAY-DECL]
$$\frac{\left\langle AD, env''_A, sto \right\rangle \rightarrow_{AD} (env'_A, sto)}{\left\langle \textbf{array}\ T\ r[n] \right\rangle \rightarrow_{AD} (env'_A, sto)}$$

where $n \in \mathbb{N}_0$
and $l_0 = env_A$ next, $l_1 = env_A$ new l_0, $l_2 = env_A$ new $l_1, \ldots, l_n = env_A$ new l_{n-1}
and $env''_A = env_A\Big[r[0] \mapsto l_0\Big]\Big[r[1] \mapsto l_1\Big]\Big[r[2] \mapsto l_2\Big]\Big[r[n] \mapsto l_n\Big]\Big[\text{next} \mapsto \text{new } l\Big]$

[PROC-DECL]
$$\frac{env_E \vdash \left\langle PD, env_P\ [p \mapsto (C, FPS, env_E, env_P)] \right\rangle \rightarrow_{PD} env'_P}{env_E \vdash \left\langle \textbf{procedure}\ p(FPS)\ \{C\}, env_P \right\rangle \rightarrow_{PD} env'_P}$$

[FUNC-DECL]
$$\frac{env_E \vdash \left\langle FD, env_F\ [f \mapsto (C, T, FPS, env_E, env_F)] \right\rangle \rightarrow_{FD} env'_F}{env_E \vdash \left\langle \textbf{function}\ T\ f(FPS)\ \{C\}, env_F \right\rangle \rightarrow_{FD} env'_F}$$

where $env_E = env_V, env_C, env_A, env_P$

Table A.4: Transition rules for declarations.

A.5 Type Rules for Arithmetic Expressions

The type rules for arithmetic expressions are shown in Table A.5.

[ADD]
$$\frac{E \vdash e_1 : T\ E \vdash e_2 : T}{E \vdash e_1 \texttt{ + } e_2 : T}$$

[SUBTRACT]
$$\frac{E \vdash e_1 : T\ E \vdash e_2 : T}{E \vdash e_1 \texttt{ - } e_2 : T}$$

[MULTIPLY]
$$\frac{E \vdash e_1 : T\ E \vdash e_2 : T}{E \vdash e_1 \texttt{ * } e_2 : T}$$

[DIVIDE]
$$\frac{E \vdash e_1 : T\ E \vdash e_2 : T}{E \vdash e_1 \texttt{ / } e_2 : T}$$

[MODULO]
$$\frac{E \vdash e_1 : T\ E \vdash e_2 : T}{E \vdash e_1 \texttt{ \% } e_2 : T}$$

Continued on the next page...

[PARENTHESIS]	$\dfrac{E \vdash e_1 : T}{E \vdash (e_1) : T}$
[NUMERAL]	$E \vdash n : T$
[VARIABLE]	$\dfrac{E(x) = T}{E \vdash x : T}$
[CONSTANT]	$\dfrac{E(c) = T}{E \vdash c : T}$
[ARRAY]	$\dfrac{E(r) = T}{E \vdash r : T}$

Table A.5: Type rules for arithmetic expressions.

A.6 Type Rules for Boolean Expressions

The type rules for boolean expressions are shown in Table A.6.

[EQUALS]	$\frac{E \vdash e_1 : T \; E \vdash e_2 : T}{E \vdash e_1 \texttt{ == } e_2 : \texttt{bool}}$
[NOTEQUALS]	$\frac{E \vdash e_1 : T \; E \vdash e_2 : T}{E \vdash e_1 \texttt{ != } e_2 : \texttt{bool}}$
[GREATERTHAN]	$\frac{E \vdash e_1 : T \; E \vdash e_2 : T}{E \vdash e_1 \texttt{ > } e_2 : \texttt{bool}}$
[GREATERTHANEQUALS]	$\frac{E \vdash e_1 : T \; E \vdash e_2 : T}{E \vdash e_1 \texttt{ >= } e_2 : \texttt{bool}}$
[LESSTHAN]	$\frac{E \vdash e_1 : T \; E \vdash e_2 : T}{E \vdash e_1 \texttt{ < } e_2 : \texttt{bool}}$
[LESSTHANEQUALS]	$\frac{E \vdash e_1 : T \; E \vdash e_2 : T}{E \vdash e_1 \texttt{ <= } e_2 : \texttt{bool}}$
[NOT]	$\frac{E \vdash e_1 : \texttt{bool}}{E \vdash \texttt{!} e_1 : \texttt{bool}}$
[AND]	$\frac{E \vdash e_1 : \texttt{bool} \; E \vdash e_2 : \texttt{bool}}{E \vdash e_1 \texttt{ \&\& } e_2 : \texttt{bool}}$
[OR]	$\frac{E \vdash e_1 : \texttt{bool} \; E \vdash e_2 : \texttt{bool}}{E \vdash e_1 \texttt{ \|\| } e_2 : \texttt{bool}}$
[PARENTHESIS]	$\frac{E \vdash e_1 : \texttt{bool}}{E \vdash (e_1) : \texttt{bool}}$

Table A.6: Type rules for boolean expressions.

A.7 Type Rules for Commands

The type rules for commands are shown in Table A.7.

[VAR-ASS]	$\frac{E \vdash x : T \; E \vdash e : T}{E \vdash x = e : \text{ok}}$
[ARRAY-ASS]	$\frac{E \vdash x[e_1 : \texttt{int}] : T \; E \vdash e_2 : T}{E \vdash x[e_1] = e_2 : \text{ok}}$

Continued on the next page...

[IF]	$\dfrac{E \vdash e : \mathbf{bool}\ E \vdash C : \mathrm{ok}}{E \vdash \mathbf{if}(e)\ \{C\} : \mathrm{ok}}$
[IF-ELSE]	$\dfrac{E \vdash e : \mathbf{bool}\ E \vdash C_1 : \mathrm{ok}\ E \vdash C_2 : \mathrm{ok}}{E \vdash \mathbf{if}(e)\ \{C_1\}\ \mathbf{else}\ \{C_2\} : \mathrm{ok}}$
[WHILE]	$\dfrac{E \vdash e : \mathbf{bool}\ E \vdash C : \mathrm{ok}}{E \vdash \mathbf{while}(e)\ \{C\} : \mathrm{ok}}$
[CALL-FUNC]	$\dfrac{E \vdash f : T_0, (FPS : T^* \to \mathrm{ok})\ E \vdash APS : T^*}{E \vdash f(APS) : \mathrm{ok}}$
[CALL-PROC]	$\dfrac{E \vdash p : (FPS : T^* \to \mathrm{ok})\ E \vdash APS : T^*}{E \vdash p(APS) : \mathrm{ok}}$

Table A.7: Type rules for commands.

A.8 Type Rules for Declarations

The type rules for declarations are shown in Table A.8.

[VAR-DECL]	$\dfrac{E\left[x \mapsto T\right]}{E \vdash T\ x : \mathrm{ok}}$
[CONST-DECL]	$\dfrac{E\left[c \mapsto T\right]\ E \vdash e : T}{E \vdash \mathbf{constant}\ T\ c = e : \mathrm{ok}}$
[ARRAY-DECL]	$\dfrac{E\left[r[0 \ldots n] \mapsto T\right]\ E \vdash n : \mathbf{int}}{E \vdash \mathbf{array}\ T\ r\ [n] : \mathrm{ok}}$
[PROC-DECL]	$\dfrac{E \vdash \left[p \mapsto (FPS : T^* \to \mathrm{ok})\right]\ E \vdash C : \mathrm{ok}}{E \vdash \mathbf{procedure}\ p(FPS)\ \{C\} : \mathrm{ok}}$
[FUNC-DECL]	$\dfrac{E \vdash \left[f \mapsto T, (FPS : T^* \to \mathrm{ok})\right]\ E \vdash C : \mathrm{ok}}{E \vdash \mathbf{function}\ T\ f(FPS)\ \{C\} : \mathrm{ok}}$

Table A.8: Type rules for declarations.

APPENDIX B

STDIO LIBRARY

Listing B.1 shows the `StdIO` library for KAL.

```
1  # Standard input/output functionality library for KAL.
2
3  # Prints a newline.
4  procedure PrintNewLine()
5  {
6     WriteChar(ToChar(13)); # Carriage return.
7     WriteChar(ToChar(10)); # Line feed.
8  }
9
10 # Prints an integer followed by a newline.
11 procedure PrintInt(int i)
12 {
13    WriteInt(i);
14    PrintNewLine();
15 }
16
17 # Prints a character followed by a newline.
18 procedure PrintChar(char c)
19 {
20    WriteChar(c);
21    PrintNewLine();
22 }
23
24 # Prints a float followed by a newline.
25 procedure PrintFloat(float f)
26 {
27    WriteFloat(f);
28    PrintNewLine();
29 }
30
31 # Prints a boolean followed by a newline.
32 procedure PrintBool(bool b)
33 {
34    WriteBool(b);
35    PrintNewLine();
36 }
37
38 # Prints an array of characters.
39 procedure PrintString(array char string[])
40 {
41    int size;
42    size = SizeOf(string[]);
43    int counter;
44    counter = 0;
45
46    while(counter < size)
```

```
47     {
48        WriteChar(string[counter]);
49        counter = counter + 1;
50     }
51
52     PrintNewLine();
53  }
```

Listing B.1: `StdIO` library for KAL.

BIBLIOGRAPHY

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[Hü10] Hans Hüttel. *Transitions and Trees*. Cambridge, 1st edition, 2010.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 2nd edition, 1999.

[MMMNS00] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object-Oriented Analysis & Design*. Marko Publishing ApS, 1st edition, 2000.

[Seb09] Robert W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, 9th edition, 2009.

[Und09] Studieordning for bacheloruddannelsen i software. `http://www.sict.aau.dk/digitalAssets/3/3331_softwbach_sept2009.pdf`, 2009.

[WB00] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java*. Prentice Hall, 1st edition, 2000.

This page is left blank for the purpose of containing the attached CD-ROM.