Aalborg University

The Faculties of Engineering, Science and Medicine
Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg East
Phone: +45 96 35 86 00
http://cs.aau.dk

Title:
Imp: An Implicitly Parallel Language

Topic:
Language Technology

Project Period:
February 2nd - May 29th, 2009

Project Group: s404a
Anton Møller Bentzen
Frederik Højlund
Jesper Kjeldgaard
Mathies Grøndahl Kristensen
Jens-Kristian Nielsen
Simon Stubbben

Supervisor:
Lone Leth Thomsen

Number of appendixes: 4

Total number of pages: 100

Number of pages in report: 79

Number of reports printed: 8

Abstract:

Today, systems with multiple processing units are becoming more common. To fully utilize the potential of these systems, programmers have to take parallelism into consideration.
This report focuses on how to move the responsibility of parallelism from the programmer to the language processor. To realize this, we analyze the various aspects of compilers and parallel computing. In order to achieve implicit parallelism, we have defined a new programming language called Imp. We have defined semantics and a Context Free Grammar in Extended Backus-Naur Form for Imp. The implementation of Imp consists of a tokenizer, parser, checker, dependence analyzer and code generator, which are described in detail in this report.
Furthermore we have reflected on potential features of Imp if we had additional time to continue the development. We have concluded that it is possible to move the responsibility of parallelism because our product is able to implicitly parallelize a `Foreach`-loop from Imp to Java.

Preface

This report is written as a fourth semester project with the theme ”Language Technology”, by six Software Engineering students from Aalborg University, Denmark. The report is written over four months during spring semester of 2009.

Sources are listed in the bibliography and references throughout the report refer to this list. Figures and tables are numbered in accordance with their containing chapter: the first figure in Chapter 1 is numbered 1.1, the second 1.2 and so forth. Explanatory text is written underneath figures and tables. Furthermore, appearances of the word he refers to he/she.

To fully understand this report, it is expected that the reader has knowledge of computer topics similar to the level of a student that has finished the third semester of Software Engineering.

The product for this project is a language definition, translator and a brief manual. The language definition and the manual are included in the appendix and the translator source code will be attached on a CD. The CD also contains online references and documentation generated in Doxygen.

Signatures

Anton Møller Bentzen	Frederik Højlund
Jesper Kjeldgaard	Mathies Grøndahl Kristensen
Jens-Kristian Nielsen	Simon Stubbben

Report Outline

The report consists of five chapters. The following briefly describes the content of these chapters:

- Chapter 1, Introduction
The first chapter introduces the project and presents the initiating problem.

- Chapter 2, Analysis
This chapter contain important general information about development of new languages and compilers for them, as well as parallel computing theory and dependence analysis.

- Chapter 3, Design
This chapter deals with thoughts and considerations of the language and compiler construction.

- Chapter 4, Implementation
The implementation explains how the translator was developed.

- Chapter 5, Recapitulation
The last chapter in the report finalizes and sums up the project.

Approach

Choosing the right development method for a software project is an important part of the development process. Instead of following one particular development method, we plan to mix practices from various methods to form our own work routine. The practices are chosen based on what the team members have had good experiences with in previous projects and some new practices that we wish to try out together.

Standing Daily Meetings

We aim to start each day with a short meeting where all attendants stand up. This meeting should provide a better overview of the finished and upcoming tasks and how the project is coming along. Members stand up during the entire meeting to keep it as concise as possible. Each meeting will proceed from the following agenda:

1. What was done the previous workday?

2. Is the work proceeding according to plan?

3. Which tasks should be done today, and by whom?

The meeting will start with a brief update on what was done the previous workday. This is to ensure that the team is working in the right direction, at the right pace, and that all members agree on the overall project status. When the work of the previous day has been discussed, the tasks for the present day will be talked about. This will produce an agenda for the day, so all team members know what is currently being worked on and what should be worked on next.

Week Encapsulating Meetings

In the same spirit as the standing daily meetings, we plan to have a longer meeting Monday morning and Friday afternoon. The Monday morning meeting focus on what we would like to achieve for the coming week, in terms of finished tasks. At the meeting Friday afternoon we look at the minutes from the Monday meeting to see if we accomplished the things we set out to do. If we achieve our goals for that week, we will consider if we over- or underestimated the tasks to ensure an optimal project velocity. If the goals are not achieved, we will investigate if this was due to the tasks being harder than first assumed, or if it is due to other problems.

Software Testing

Two kinds of testing is going to be used in this project: User and correctness testing. The user testing will be performed throughout the development phase to check if the implemented tasks function as intended. The correctness tests will be executed in the last stage of the development to ensure that the program does what it should in terms of correct output.

We chose not to use unit test in this project because few in the group have used it before and therefore it will be a time consuming practice to use.

Pair Programming

We will use the pair programming practice from eXtreme Programming (XP) during the development phase[2]. Pair programming increases the quality of the code hereby reducing the amount of possible bugs. Furthermore, it helps the team share their knowledge. When learning new material it is often easier to understand it if there is a fellow developer to discuss the material with.

Pair programming will be used because of the benefits described above, but also because we have had good experiences with it in previous projects.

Milestone-Driven Development

To have some idea of how far along the project is, we will create milestones for various stages in the development process. A milestone contains a formal definition of an expected state the project and/or the product should be in at a certain date. We think using milestones is a good way to ensure that the overall deadline can be reached by splitting it into smaller intervals with separate deadlines. Each milestone contains all the tasks that are required to be completed in order to reach the milestone. This will also be the primary way of measuring when these are done.

Project Management Support Tool

During the previous semester, we learned that using a project management support tool was a great way to keep track of tasks in the project. Therefore we plan to use Trac to manage tasks and milestones in this project. We have never used this tool consistently in a project, only tested it as part of a course. After reviewing the various solutions available, we will use Trac because it focuses on milestones and tasks which suit our needs very well.

Using a web based tool like Trac makes it easy for team members to manage the tasks they are assigned to from home. Furthermore the entire team has access to the tool making it easy to get information on the project status and which members are working on specific tasks. A possible problem using Trac is that all team members must use it consistently for it to have any value. Using Trac will therefore require all members to put an effort in maintaining their respective tasks.

Table of Contents

Chapter 1
Introduction

In 1965 Gordon E. Moore defined a law stating that the number of transistors placed on integrated circuits would double every two years[17]. Today, Moore's law differs from reality for processing power due to power dissipation issues in CPU architecture design. As a result of this, more cores are being fitted onto CPUs, hereby reducing power consumption whilst improving performance. This change in CPU architecture imposes a challenge for the developers to utilize multiple processing units, which should give the user a boost in performance.

Implicit parallelism is a mechanism that makes it possible for a developer to utilize the multiple processing units while still writing sequential code. That is, a language providing implicit parallelism assures that a source program is made to run concurrently by a language processor, without explicitly dividing the program into independent parts, even though the program was written with sequential statements. Implicit parallelism is further explained in section 2.5.

The developers who are utilizing multi-core processors, will have to learn a language that supports concurrent or parallel programming (a difference between these is given in section 2.4.4). Many of the general purpose languages include constructs for parallel programming, but using them could reduce productivity of the developer. Therefore we need to remove the task of making programs concurrent from the developer to another part of the software development process. This part could be a compiler that automatically translates the source program into a program that utilizes multi-core processors, as with implicit parallelism. This leads to the derivation of the initiating problem for this project.

Initiating Problem

As written above, the general subject in this project is to make use of parallelism and make the computer exploit it automatically. From this theme, the following questions have been constructed as motivation for the project:

- Is it possible to automate the process of creating parallelism for a given source code?
- How can we automate the task of translating sequential code into parallel code, hereby removing the responsibility of writing parallel code from the developer?
- Can a language processor assist the developers when writing parallel programs?

Through the following chapters, our decisions for the development process are taken into account and the alternatives and consequences of the choices are considered.

Chapter 2 Analysis

Constructing a programming language from scratch is a demanding and time consuming task, so before starting to design and implement the implicit parallel language, an analysis of relevant aspects is needed. This chapter is split into eleven sections concerning these aspects.
Knowledge about how a compiler is constructed manually is explained in section 2.2, whereas tools for automating the process is briefly discussed in the following section. As written in the Introduction, this project is about parallel computing, therefore this subject as well as the implicit part is worked out in sections 2.4 and 2.5. Next, section 2.6 goes through theoretical issues regarding dependence, which we think is an important phase of parallelism. To get a general understanding of implicit parallelism, section 2.7 provides a general overview of some of the existing languages and compilers. Choice of development language, target language and criteria for the language to be developed are explained in sections 2.9 and 2.10. Finally, the Problem Statement in section 2.11 narrows down the initial problem and the Summary (section 2.12) closes the analysis chapter.

2.1 Language Processors

To run a computer program on a machine, the program must be accepted by the set of operations defined on the target machine. A programming language processor is a system that performs translation or interpretation of a program written in a given programming language (source language) into another language (target language). This section will cover the basics and differences of translation and interpretation, a more in depth description of a translator is described in the next section (2.2). Using language processors it is possible to run programs or make it easier to run them on different types of machines.

The following cases of translation take a higher level language A as input and outputs a lower level language B, see figure 2.1. In this section a high level language is defined as languages designed to be used by application programmers with problems to solve, levels 5 and 4 according to figure 2.1. The reason for calling assembly high level (level 4) is because of its use of words and abbreviations. A low level language is written by system programmers and is very close to the hardware, levels 3 and below on figure 2.1. Low level languages are characterized by being difficult to use for writing advanced programs [23].

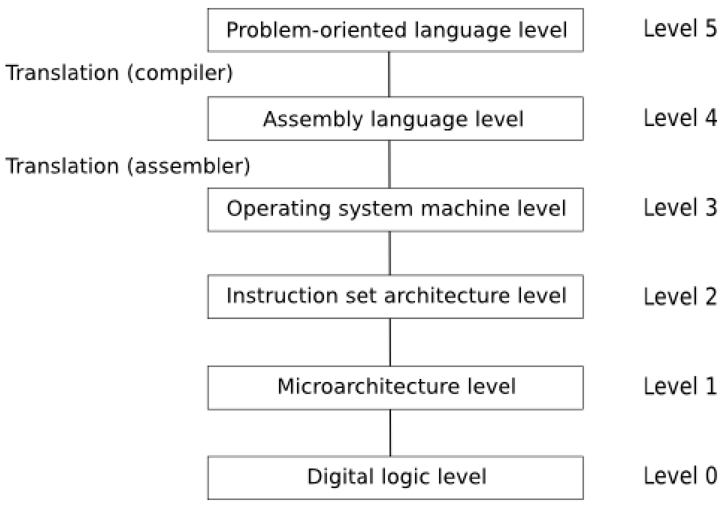


Figure 2.1: A six-level computer model.

2.1.1 Translation

A translator translates source code from language A to language B. The translation processor takes in a language A, which is accepted by the translator, and generates source code in the target language B that is semantically equivalent, see figure 2.2. Translation between two equally leveled languages is also possible and is often called source-to-source translation.

Assemblers and compilers are two types of translators. An assembler is a translator that translates from assembler code to machine code, whereas a compiler is defined as a processor that translates from a level 5 language to a lower level language, usually level 4 or 3, see figure 2.1. C and C++ are examples of languages that use compilers.

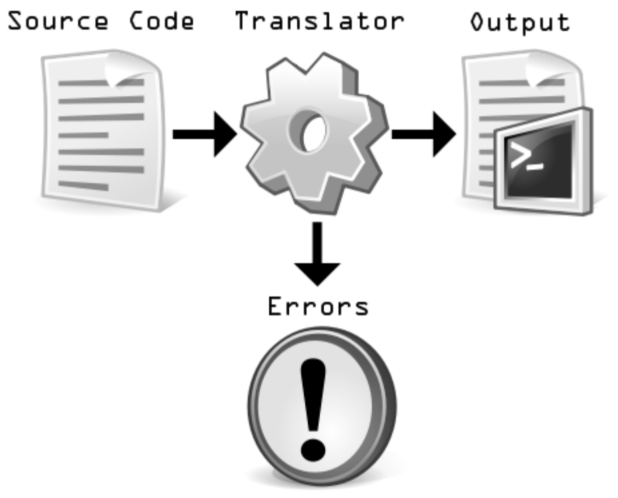


Figure 2.2: Blackbox model of a compiler

It can be a complex task to write a compiler by hand, which is why compiler-compilers are useful. A compiler-compiler is a tool that generates compilers or interpreters from a formal description. This process and some of these tools are looked at in section 2.3. A language processor that does the reverse, translating the language from low to high level, is called a decompiler [7]. Decompilation can be useful when recovering lost source code or checking a program for vulnerabilities for security purposes, because higher level languages are easier for the programmer to understand.

2.1.2 Interpretation

The interpreter works by going through small parts of the source language A and immediately executes the result on the machine. That is, the interpreter fetches, analyzes, and executes the source program instructions one at a time, see figure 2.3. The portability of interpreted programs is increased, as the only matter is, in theory, the interpreter's compatibility with the given machine. The interpretation process can be slower than translation because it fetches, analyzes and executes each instruction, which is very poor when done frequently or when instructions are very complicated. Some interpreters solve this issue by including a Just-In-Time (JIT) compiler that translates frequently used code into machine code and hereby optimizing the performance of the program.

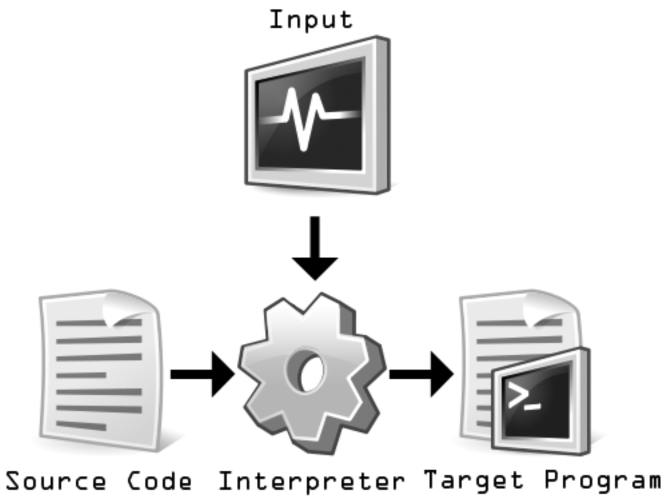


Figure 2.3: Blackbox model of an interpreter

2.2 Compilation Phases

This section contains a brief, general description of the syntactical, semantical, contextual and code generation phase of processing a programming language. To implement a language, these phases are necessary to be able to determine how the elements of the language work. The syntax of a programming language is a structure consisting of expressions, statements and program units. Semantics, on the other hand, is the meaning of this structure. Furthermore, this section will include terms like a "context-free grammar" (CFG) and "parser" which will be elaborated. The following is based on the sources [22] and [24].

2.2.1 The Syntactical Phase

The syntactical element of a programming language consists of a set of rules which defines combinations of symbols that are considered to be syntactically correct. The lexical structure of a programming language is usually defined by regular expressions, while the grammatical structure is written as a CFG in Backus-Naur Form (BNF) to define terminal and non-terminal symbols. We are able to formally define a programming language in terms of a CFG, which consists of the following elements:

- **Terminal symbols** are the atomic symbols which make up the elements that are used in the actual language. Examples of typical terminal symbols in a programming language are ';', 'while', 'if' and 'else'.
- **Non-terminal symbols** are the presentation of specific classes of phrases in the language. Examples of typical non-terminal symbols in a programming language are Program, Command and Expression
- A **start symbol** is defined as being the primary class of the available phrases in the programming language. The start symbol itself is a non-terminal, and is often defined in the programming language's grammar as Program.
- **Production rules** are needed in order to compose phrases from terminal symbols and sub-phrases. These are needed in order for the parser to construct a tree from the terminal and non-terminal symbols. Such a tree is called an Abstract Syntax Tree (AST), and is described further below.

A set of production rules written in Extended BNF (EBNF) is exemplified below:

Listing 2.1: Example of an EBNF

```
Command ::= Single-command
                    |    Single-command Command

Single-command   ::=      Name (( ( Function-call-parameterlist )
   | = Expression ));
          |   if ( Bool-expression ) { Command } [else { Command
              }]
          |   while ( Bool-expression ) { Command }
          |   { Command }
          |   Declaration
          |   Returner ;

Expression       ::= Term (( (( + | - )) Term ))*
```

A set of production rules written in (E)BNF can also contain information like names for production rules etc.

The main objective of a syntactical analysis is to make use of a parsing algorithm, parse the source program and thereby determine its phrase structure. When parsing, it is convenient to consider the source program as a stream of tokens in order to find the terminal and non-terminal symbols. These tokens are symbols such as identifiers, literals, operators etc. which the parser uses to build up the AST. Since the source of a program consists of characters, and a token may consist of several characters, it is necessary to scan through the source in order to tokenize the terminal and non-terminal symbols. Splitting up the source code into tokens is done by the tokenizer, also referred to as the lexical analyzer.

Parser Classes

Parsers are categorized in classes according to how they work. The different parser classes can be seen in figure 2.4.

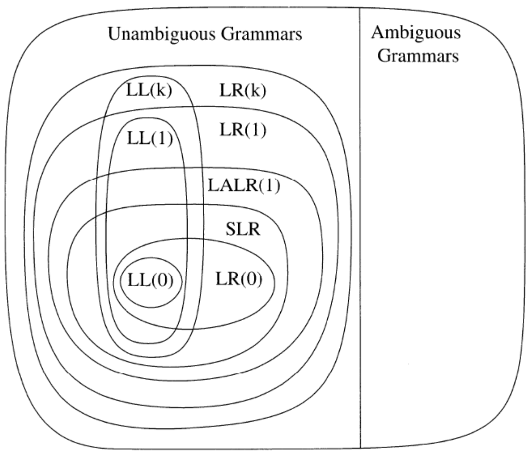


Figure 2.4: Relationship between the grammar classes. This figure is from the slides from the SPO course.

An LL parser uses top-down parsing. It is called an LL parser because it parses from Left to right, and constructs a Leftmost derivation of the sentence. An LL parser first finds each non-terminal production which can be started off with, called the start set. Afterwards, it begins with the start symbol and compares the start sets of the different productions with the first piece of input to determine which productions can be used.

The tree is built from the root to the leaves (usually trees in computer science have their root in the top and their leaves in the bottom), which is why it is also called top down parsing. The *k* defines the maximum look-ahead of the parser, meaning how far ahead in the production rules the parser looks when trying parse the program [9].

LR parsers use bottom-up parsing, which means the tree is built from its leaves to its root. It reads input from Left to right and produces a Rightmost derivation, hence LR parsing. This technique collects input until it can reduce an input sequence to a symbol.

A SLR or Simple LR parser is created by an SLR parser generator that can read a BNF grammar and construct an LR(0) parser and computes the look-ahead sets for each reduction in a state by examining the follow set for the non-terminal associated with the reduction. The Follow Set for each non-terminal symbol is computed by seeing which terminal symbols follow the non-terminal symbol in the rules of the BNF grammar.

A LALR Parser or a Look-ahead LR parser, is a variant of a LR parser. LR Parsing combines related ”configuration sets” thereby limiting the size of the parse tables [6].

Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree representation of the syntactical structure of the source program and is built by the parser from a given list of tokens. The parser uses the production rules from the CFG to define the structure of the tree, meaning how the tokens are organized. An example of an AST is shown in figure 2.5.

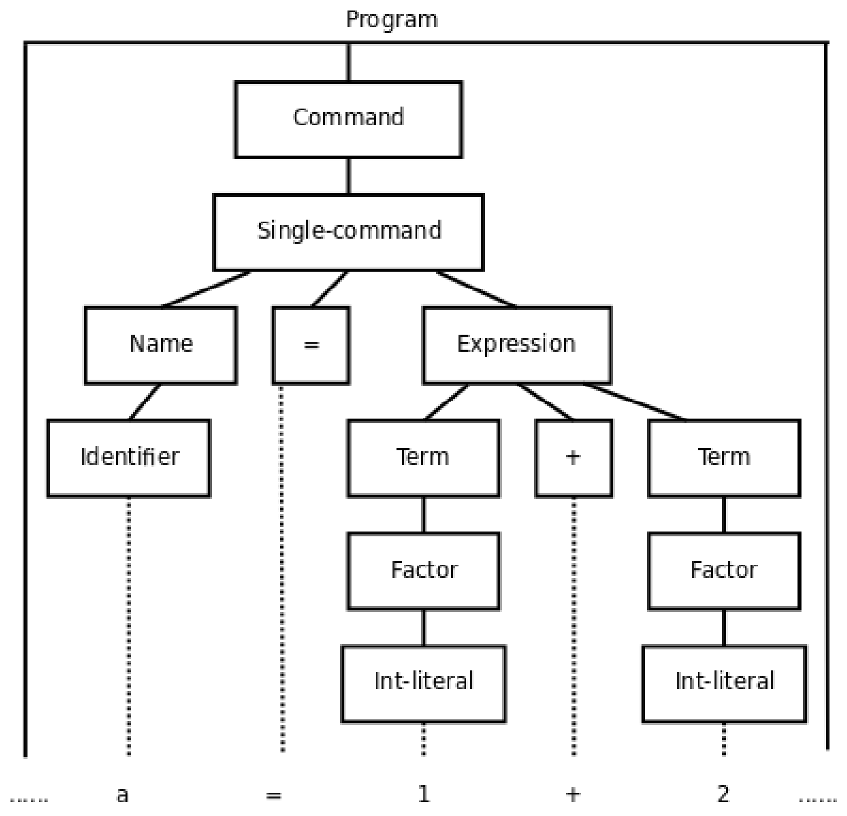


Figure 2.5: Abstract Syntax Tree Example

2.2.2 The Contextual Phase

In order to describe the behavior of programs, the semantics of the programming language needs to be defined. Standards for describing semantics have been developed through time and referred to as formal semantics. Another method of defining the semantics is by describing it in an informal language. Formal and informal semantics are elaborated in the following.

Formal Semantics

The most commonly used formal standards are operational, denotational and axiomatic semantics. The operational semantics (OS) of a program are described by means of transitions from one program state to another. OS cover two types of semantics, namely Big-step and Small-step semantics. Big-step semantics describe the entire transition, from a configuration to an end-configuration, in one (big) step. Small-step semantics describe a transition, from a configuration to an end-configuration, through a series of middle-configurations; in other words step by step. Denotational semantics (DS) define a denotation or meaning of a program in a function, which maps input into output. Denotational semantics is formal in describing a programming language, as mathematical objects denote the meaning of its corresponding syntactical element. In axiomatic semantics (AS) a language is described by mathematical logic, typically by setting rules for which logical formula that applies before and after the execution, that is to prove the correctness of the program. [12]

- Operational semantics tell *how* a program is executed.
  - Big-step semantics describe a transition in one step.
  - Small-step semantics describe a transition in multiple steps.
- Denotational semantics tell *what* a program does.
- Axiomatic semantics tell *why* a program behave in a certain way.

Informal Semantics

Informal semantics describe the semantics of a programming language by using an informal ”real-world” language. As opposed to formal semantics, there are no commonly agreed on standards for informal semantics. In a way, informal semantics can be compared to pseudo-code where there is no strict definition. The main advantage of using an informal semantic is that they are easy to comprehend by the reader, hereby possibly reaching a larger audience. The drawback of informal semantics is that that they lack the concise expressiveness of formal semantics. This means that whereas informal semantics might derive multiple implementations due to its vagueness, formal semantics only lead to one implementation.

Semantics in General

The main objective of both semantical types is generally to specify the semantics of the production rules (our classes of phrases) in the language. For instance, it is necessary to specify the semantics of Command, Single-Command and Expression in order to provide the meaning of these and to ensure they appear in the right context even though they might be syntactically correct. Before describing the various steps of the contextual analysis we will introduce the concepts of static and dynamic typing.

Static and Dynamic Typing

If a programming language is typed statically, then all expressions must have their types determined at compile-time, prior to the execution of the program. There are two options for static typing:

1. The first option is to make the language manifestly typed. If this is the case, the programmer has to explicitly declare variables to be of a certain type.
2. The second option is to make the language type-inferred. This means the compiler will infer the types of the expressions at compile time based on the context.

However, if a programming language is typed dynamically, type checking is performed at run-time. Types are determined based on the values of the variables, rather than based on their declarations. Dynamic typing is generally more flexible than static typing, but it is also more prone to having type errors, since one does not necessarily have any guarantee that the input will have the correct type.

Steps of the Contextual Analysis

The program is analyzed to determine if the source code satisfies the following contextual constraints:

- Assuming the source language is statically typed, the source code will be checked to see if the scope rules of the language are fulfilled. This makes it possible to detect undeclared identifiers at compile-time.
- Type rules ensure that it is possible to detect type errors and infer the type of each expression at compile-time. Furthermore, variable declarations and references are type-checked to ensure that the source program does not contain ill-formed assignments, e.g. assigning a string to an integer. This also assumes the use of static typing in the programming language.

During the contextual analysis, the contextual constraints are applied to the AST which results in a decorated AST. The scope rules are applied by linking the applied occurrences of an identifier to its corresponding declaration, while the type rules additionally decorate the abstract syntax tree. In short, if the source program satisfies the contextual constraints, the result is a decorated AST, if not, the compilation process cannot continue and an error will be printed.

2.2.3 Code Generation

Generating code is the final task of the compiler or translator. By now, the source code has been checked for syntax and contextual errors and if none are detected, code can be generated. The code is generated based on the source code and the semantics of the source program[24].

An issue when a compiler is generating target code is to make sure that identifiers are bound with the values they have been declared with. For example:

Listing 2.2: const declaration

```
    const int x = 42;
```

Given the code in listing 2.2 the code generator of a compiler can replace occurrences of the identifier x with 42, and then generate appropriate machine or assembly code. The code generator must generate code with the semantics of the source language and with the syntax of the target language.

2.3 Parser Generators

Sometimes writing a parser by hand is unnecessary because it can easily be generated by the use of a parser generator. A parser generator is a software tool, that generates source code for a parser. The parser generator takes input in the form of the grammar of a programming language, usually in BNF. Most parser generators generate both a tokenizer (lexical analyzer) and a parser. A lexical analyzer divides the input text into recognizable keywords, or tokens, and the parser generates the abstract syntax tree. In the following sections we will look at two different parser generators, CUP and JavaCC, and one lexical analyzer called JLex.

2.3.1 JLex

JLex generates a lexical analyzer, which translates text input into tokens. JLex is based on it's predecessor Lex, another lexical analyzer generator for the UNIX operating system. Lex takes a specially formated grammar, which, among other things, contains detailed information for a tokenizer behavior. It then generates the source code in C for such a tokenizer[3]. JLex, on the other hand, generates the source code in Java instead of C.

2.3.2 CUP

CUP is software tool for generating a LALR parser given some specifications. CUP is written in Java, uses specifications written in Java-like code and it generates source code in Java. CUP takes a specially formated grammar and a tokenizer for generating a parser. This parser can determine the phrase structure of the source code, or report syntax error depending on the grammar. [11]

2.3.3 JavaCC

JavaCC is another software tool that can provide both lexical and syntactical analysis. With JavaCC you specify a grammar similar to an EBNF, which is advantageous if you already have a grammar written in EBNF. With JavaCC comes JJDoc and JJtree. JJDoc generates documentation in HTML for the given grammar, similar to Javadoc. JJTree generates actions that build a tree structure when parsing source code for a new program. It provides a framework based on the Visitor design pattern that allows traversal of the parse tree in memory. [8]

2.4 Parallel Computing

As mentioned in the introduction of the report, parallel computing has moved from large super computers into the living room due to the introduction of multi-core processors. It is this move that is driving the paradigm shift of application programming from sequential programs to parallel programs. In this section we discuss parallelism, some of the different types of parallelism and how it can be used when developing a program. The source of the following is [22].

This new, in the sense of application development, parallel paradigm requires more planning and thought from the developer as data can suddenly be accessed simultaneously. Parallelism is often referred to as concurrency or vice versa as they have a lot in common. Nevertheless, their inequality will be explained in section 2.4.4. Parallelism is divided into several levels, denoted as types in this section, but only two are looked at in this report. The two types are: data parallelism and task parallelism. For a dependence analysis of parallelism, see section 2.6.

Before getting in depth with parallel levels, a broad knowledge of parallelism is beneficial. Concepts of parallelism and its types will be described.

2.4.1 Concepts

Tasks and synchronization are two concepts that must be introduced before parallelism can be described.

A *task* is a unit of a program, like a subprogram and also referred to as a process, that is able to be executed at the same time as other units. Tasks communicate with other tasks in order to synchronize execution paths to prevent errors.

Synchronization controls the order in which the tasks are executed. Competition synchronization, which is one kind of synchronization, prevents two tasks of accessing the same data. When two or more tasks access the same data it can lead to what is called a ”race condition”. In a race condition the execution flow can change depending on what thread changes data first, for example:
Two tasks, `A` and `B`, are using a shared variable called `sum` with the starting value of 6. Task `A` adds 2 to `sum` and task B multiplies by 3. Figure 2.6 shows this phenomenon, where the value of `sum` is different depending on which task is executed first. The absence of competition synchronization here would lead to race conditions, where the execution speed of each statement defines the output of the program.

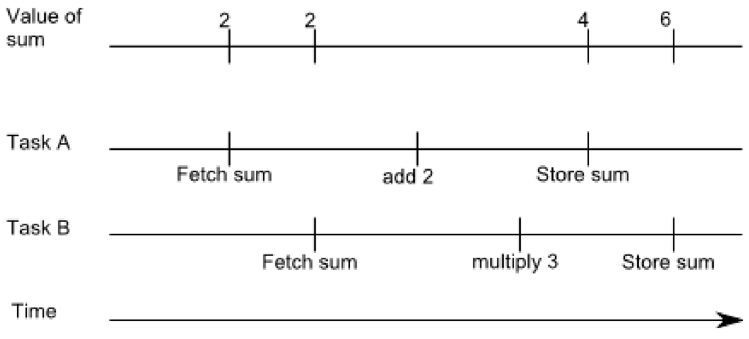


Figure 2.6: Showing the need for competition synchronization

As mentioned above several types of parallelism exist, but only two are looked at in the following. For analysis of parallelism, see section 2.6.

2.4.2 Data Parallelism

Data parallelism, or loop-level parallelism, is the concept of distributing independent data to be processed on separate execution processes (threads), whether that is on the same CPU or not is irrelevant. The term ”loop-level parallelism” originates from the fact that it is most often loops that are processed by means of data parallelism. It is not all data that can be parallelized in such a way, as order of evaluation must sometimes be enforced to produce correct output.

2.4.3 Task Parallelism

Another type of parallelism is task parallelism that runs different tasks in parallel. The tasks do largely different calculations on some data, which they may or may not have in common. An example of such parallelism can be two tasks, or functions, that calculate the sum and minimum values respectively on an array of numbers.

2.4.4 Concurrency and Parallelism

The purpose of this section is to be able to distinguish between the two terms concurrency and parallelism.

Concurrency and parallelism are two different things, but the concepts are often used indiscriminately. Given two tasks, T1 and T2, they are said to be concurrent if the execution order of the two transactions is not pre-determined in time.[13]

This means that:

- T1 may be executed and finished before T2.
- T2 may be executed and finished before T1.

Concurrency is often referred to as being a property of a software program and is a more general concept than parallelism. Concurrency programming is a term used to indicate potential parallelism in a software application.

If the two transactions, T1 and T2 were to be parallel, then the following would be true:

- T1 and T2 may be executed simultaneously at the same instance of time.

If two transactions were to share the CPU time on a single core, this would make the transactions run concurrently. On the other hand, if two transactions were to split up on two (or more) cores and be independent of one another, this would mean that the two transactions were executed in parallel.[25]

It is important to note that the previous statements are not valid when speaking of concurrent programming and parallel programming. The two areas are interestingly enough overlapped, but neither is a superset (or a subset) of the other. The reason why this is true is because the two ways of programming cover different areas of software development.

In this report we will not use these two terms in their strictest sense because focus is on the process of translating sequential code into parallel code implicitly.

2.4.5 Approaches

Parallel programming can be achieved in multiple ways. The following describes some of the different methods to exploit parallel programming[4].

Automatic Program Parallelization

In this approach, the compiler analyzes the code and extracts parallel loops. The programmer abstracts from parallelism and sequential code is converted to parallel code. As it is done automatically, the developer does not need to be involved or even worry about it. This is referred to as implicit parallelism, which is elaborated in the next section (2.5).

Explicitly Specified Parallelism

Another approach is to explicitly specify in the source code where the program is able to be divided into threads. This task is to be done by the developer hereby increasing the workload for writing parallel programs. The problem with this approach, is that it may cause new series of errors related to synchronization defects, which can be problematic to debug.

Parallelization through Speculative Execution

This approach include Thread Level Speculation (TLS) and Transactional Memory. Their key ability is to execute parts of code in a ”sand box”, where updates are buffered, and all memory access is examined for conflicts among concurrent threads. TLS can be implemented in hardware or compiler support to extract speculative tasks automatically. However, hardware support for speculation can be quite complex and is usually less efficient than explicitly parallel programs.

2.5 Implicit Parallelism

As written in the section above, parallel systems have become quite common today. However, programmers seldom exploit the extra power of the users’ modern hardware, as parallel programming is very complex compared to traditional sequential programming [19]. Different approaches to parallel programming was described above, but since this project concerns the implementation of a language with implicit parallelism, this approach is selected and described in this section.

A programming language that features implicit parallelism allows a programmer to disregard worries of exploitation of parallelism, which instead is done by the language processor or runtime environment. That is, the parallelism is hidden from the programmer, who just writes standard sequential code. This lets him focus on other issues when writing code, rather than worry about dividing tasks across multiple processors - much like early high level programming languages moved the focus of people’s minds from primitive instructions, jumps and registers, to actual algorithms.

Some books refer to implicit parallelism as when a programmer must define, for example by means of a special instruction, where the program can be parallelized. This project however, sees implicit parallelism as parallelism done completely by the compiler without any special instructions from the programmer.

As mentioned in section 2.4.5, implementation of implicit parallelism is a complex job and has not been solved but for a limited set of applications (for example extensive operations on arrays). Some examples of languages and language processors that employ implicit parallelism is given in 2.7.

2.6 Dependence Analysis

In a program, two instructions may depend on each other, This means an instruction may refer to the outcome of the preceding instruction. When restructuring the program for running parallel processes, one of the most important steps is a dependence analysis, as it is used to discover dependencies between instructions. Programs may have execution order constraints in between statements because of dependencies, which is why a dependence analysis should determine the order of the statements to be able to parallelize them. Dependence analysis is to control the order of statements for optimizing the program. This section is written by means of [10] and [18] and will deal with dependence analysis covered by the following subjects:

- Data dependencies
- Control dependencies
- Dependence graphs
- Loop dependencies

2.6.1 Data Dependencies

Data dependence appears due to conflicts accessing shared data. The codelines below is an example of data dependency between statements S_3 and S_1, S_2.

```
S1:  pi = 3.14;
S2:  r = 5.0;
S3:  area = pi*r*2;
```

S_1 and S_2 are two variables used in S_3, resulting in S_3 being dependent on S_1 and S_2. This example is elaborated in figure 2.7 when describing dependence graphs.

A.J. Bernstein established a set of conditions, called Bernstein's conditions, determining whether statements are parallelizable. Bernstein's conditions are:

- $\mathrm{I}(S_1) \cap \mathrm{O}(S_2) = \oslash$
- $\mathrm{O}(S_1) \cap \mathrm{I}(S_2) = \oslash$
- $\mathrm{O}(S_1) \cap \mathrm{O}(S_2) = \oslash$

where S_i are statements, I (input) is the set of memory locations read and O (output) is the set of memory locations written. Moreover, a feasible run-time execution path from S_1 to S_2 is available. Below is an example of how to make use of the conditions:

```
S1:  a = x + y;
S2:  b = x + z;
```

where $I_1 = x, y$, $O_1 = a$ and $I_2 = x, y$, $O_2 = b$. These statements fulfill Bernstein's conditions and are therefore parallelizable.

There are different cases of dependence explained briefly in the following.

- Flow dependence:
  A statement S_2 is flow dependent on S_1 if S_2 reads data written by S_1 and S_1 precedes S_2. Example:

  ```
  S1:  a = ...;
  S2:  ... = a;
  ```

- Anti dependence:
  A statement S_2 is anti dependent on S_1 if S_2 stores data preceding a read by S_1. Example:

  ```
  S1:  ... = a;
  S2:  a = ...;
  ```

- Output dependence:
  A statement S_2 is output dependent on S_1 if both S_2 and S_1 write to the same location. Example:

  ```
  S1:  a = ... ;
  S2:  a = ... ;
  ```

2.6.2 Control Dependencies

The evaluation of a statement might depend on the result of a previous statement. That is, a statement S_2 is control dependent on S_1 if the execution of S_2 is conditionally guarded by S_1. For example:

```
S1:  if (a ¡ 0)
S2:    b = 1;
S3:  c = 2;
```

In this case S_2 is control dependent on S_1 whereas S_3 is control independent.

2.6.3 Dependence Graphs

Data and control dependencies can be visualized by graphs. A dependence graph is a directed graph that shows how statements depend on each other. Definition:

- Directed graph $G = (N, E)$ where

  - N is the set of statements (nodes)
  - (S_1,S_2) ∈ E if S_2 depends on S_1 (edges)

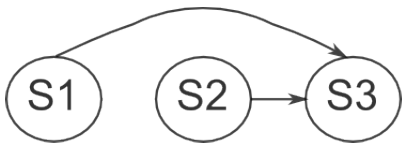


Figure 2.7: Dependence graph

Figure 2.7 is a dependence graph of the statements given in the beginning of section 2.6.1, which shows how S_3 is flow dependent on S_1 and S_2.

2.6.4 Loop Dependence

The previously discussed dependence problems can be directly applied to loops, as a loop is just a clever way of executing the same statement a number of times. Thus the loop can be analyzed the same way regular statements can. Below is an example of a loop with no dependencies.

Listing 2.3: Example of a loop with no data dependencies

```
for(int i = 1; i < 10;i++) {
a[i] = a[i] * 2;
}
```

It is fairly trivial to see that an iteration of the loop is independent of all the other (it does not use data computed in an earlier iteration). Therefore it can be divided into multiple threads without changing the actual output of the program. below is an example of a loop with data dependencies.

Listing 2.4: Example of a loop with data dependencies

```
for(int i = 1; i < 10;i++) {
a[i] = a[i-1] * 2;
}
```

The loop above has data dependencies because each iteration uses the value of the previous index in the array. The result of this is that the loop cannot be parallelized because doing so would change the output of the program, hereby changing the logic expressed by the developer.

2.7 Existing Languages with Implicit Parallelism

When starting a project for solving a problem, it is often a good idea to analyze some of the existing solutions available to get inspiration for a new solution. Implicit parallelism is a rare feature in today's programming languages, and those that offer it are often at a very experimental stage or targeted at a specific problem domain like scientific mathematics. In our opinion this is most likely because regular desktop applications, would not gain enough performance from being programmed in a language containing implicit parallelism. Two of the languages that have included implicit parallelism are Parallel Haskell (pH) and Fortress. Another project, the JAVAR compiler, restructures sequential Java code into threaded Java.

The following subsections describe some of the characteristics of these three projects with mind on what ideas we might be able to use in our project.

2.7.1 Parallel Haskell

Parallel Haskell (pH) is a parallel variant of the Haskell programming language. Regular Haskell is a purely functional programming language that has been developed for more than twenty years with focus on making it easier to produce flexible and maintainable software. Regular Haskell includes various constructs for parallel and concurrent programming, but does not include implicit parallelism.[5]

The pH compiler uses an eager evaluation model. The eager evaluation model evaluates expressions from variable assignments as soon as they get bound, whereas its counter part, lazy evaluation (used in Haskell), postpones evaluation until the variable is used. The front-end of the pH compiler is based on a Haskell compiler. By using the same front-end as Haskell, the pH compiler allows direct parallelization of existing Haskell programs. This enables Haskell developers to use the pH compiler to obtain implicit parallelism, which is favorable because it hereby creates a larger target audience for pH.[16]

2.7.2 Fortress

Fortress is a relatively new programming language, released in April 2008, being developed by Sun Microsystems. It is designed for high-performance computing and focuses on high programmability by letting the developer customize the semantics of the language to their liking. Fortress includes parallelization as default on various expressions and data structures, but sequential execution is also possible by expressing it explicitly in the program. Most of the functionality in Fortress is contained within libraries. This enables developers to change their ”version” of the Fortress language by adding their own libraries. For instance if the developer wants another syntax for the ”for” loop, he can simply create his own library and include that instead of using the default syntax.[15]

2.7.3 JAVAR

JAVAR is a restructuring compiler that translates sequential Java code into a multi threaded Java program. It focuses primarily on parallelizing loops and recursive methods. The JAVAR compiler does not include a full implementation of the Java front-end which results in some of the semantical analysis being left out. This does not matter much because the target language is Java that is being compiled again by the Java compiler that contains the full set of semantical analysis. This strategy enabled the developers of the JAVAR compiler to focus more on the main priority, implicit parallelism, and not spend much time on implementing the semantic analyzer.

2.7.4 Favorable Features

Derived from the analysis throughout this section we have come up with the following features and concepts we would like our language to contain.

- Make it easy for developers that are proficient with languages similar to ours to utilize implicit parallelism.
- Having a high-level target language enables using the semantic analyzer of that language.

2.8 Programming Paradigms

The programming paradigm for a language is important, as the paradigm decides the primary style of the languages syntax and semantics. This section will give an overview of three major programming paradigms. The paradigms are listed below with other major paradigms as well. The section is based on [21].

The three paradigms:

- Imperative
- Functional
- Object-oriented

The other paradigms are the logical, visual, constraint based and parallel paradigm (parallel is described in section 2.4 and as implicit in section 2.5).

2.8.1 Imperative Paradigm

In the imperative paradigm the order of execution is important, and the idea is based on digital hardware discipline. The command is the computational step and therefore the measurable effect on the program state. This property is well described by ”first do this, then do that”, which is similar to a cooking recipe. Typical commands of the paradigm are assignment, input/output and procedure calls. The natural abstraction is the procedure, which selects one or more actions to a procedure that can be executed as a single command. Examples of imperative programming languages are: Fortran, Pascal and C (see listing 2.5).

Listing 2.5: Hello world in C

```
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
    return 0;
}
```

2.8.2 Functional Paradigm

The functional paradigm is simple and clean, as it derives from the mathematical discipline: functional theory. Every computation is done by calling functions, evaluate the result and use it. The natural abstraction is the procedure, which selects a single expression to a function that can be calculated as an expression. Examples of functional programming languages are: Lisp (see listing 2.6), Scheme and Haskell.

Listing 2.6: Hello world in Lisp

```
(DEFUN HELLO ()
  "HELLO WORLD"
)
```

2.8.3 Object-Oriented Paradigm

The general idea of object-oriented programming (OOP) is to take real life objects and describe them in a programming language as classes. These classes contain associated data or characteristics of the given element. OOP’s support of encapsulation and logical grouping is one of the factors that have made it widely popular among developers, especially when larger programs are being developed. The object is an individual instance of a class and it encapsulates data as well as operations derived from the class. A code class is able to inherit data and functionality from another class, which means the classes are organized in hierarchies. Examples of object-oriented programming languages are: C++, Java (see listing 2.7) and C#

Listing 2.7: Hello world in Java

```java
public class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello World!");
  }
}
```

2.9 Choosing Target and Development Languages

Before we can write our translator, we need to select which programming languages to use in the process. We need a language to develop the translator in, and another language that the translator translates the source program into.

2.9.1 Development Language

We have chosen not to focus on learning a new programming language this semester, so we need to choose one we all have some experience with. It will also be an advantage to choose a language that is used as an example in some of the literature of this semester. The members of our group use different operating systems, so a cross-platform programming language will be needed to cope with this. Furthermore, we all have experience with OOP which, in our opinion, increases both flexibility and reusability, so selecting a language that supports this is advantageous.

This taken into consideration, the choice of development language is between two candidates, namely C# and Java. These are chosen, as we have had courses in both languages and have experience using them in previous projects. An argument for choosing Java is its virtual machine, the JVM. The JVM makes cross-platform development very easy and this is an advantage because we use different operating systems. The Language and Compiler Construction (SPO) course we attend this semester, is partially based on a book [24], that uses Java to describe subjects about programming languages and compilers. We believe that, C#, as opposed to Java, is not preferable when developing cross platform software. This is a controversial subject since tools do exist for writing and compiling C# code and running .NET applications on an open-source cross-platform CLR, called Mono. However our experiences using Java have been more positive, than those using C# with Mono. Another difference is that the JVM is designed for platform independence, whereas the C# runtime environment is designed for language independence. Since C# is newer than Java, literature about translators and compilers in C# is not widely available yet, hence the use of Java in the SPO course.

To recap, we found the following arguments for choosing Java as the development language:

- Experience

- Cross-platform
- Object-Oriented Programming
- Java is used in the SPO course

2.9.2 Target Language

To achieve parallelism in the compiled program, our target language should be one that supports parallel programming. This is because we have to be able to explicitly decide which processes should run concurrently. A simple way to achieve this, is by using a target language that supports threaded programming.
The main idea of multiple execution paths running concurrently will still be shown, and we will not waste time perfecting parallelism in our program output. Threaded programming is explained further in the design chapter. As with the development language, a cross-platform target language would be preferable due to our development environment. Having experience with the target language and its syntax is an advantage.

We choose Java over C# again because of it being cross-platform and used in a semester course. Furthermore, since Java is chosen as the developing language, it will be an advantage having the same target language because we only have to focus on one language, besides our product language, throughout the project.

To recap, we found the following arguments for choosing Java as the target language

- Experience
- Cross-platform
- Threaded programming
- Same as the development language

2.10 Language Evaluation Criteria

Before defining the evaluation criteria, we need to define the language's target audience and what its purpose is. The target audience for our language is students with a knowledge level equal to our own, as written in the Preface. Having finished the third semester of the Software Engineering education, the target audience have some experience with programming and programming languages. Experience with thread programming or concurrency in general is not required, and since none of the courses up until the fourth semester have dealt with these subjects, this fits well with our target audience. The purpose of the language is to run smaller programs on multiple threads and thereby increase performance.

To be able to formally define what should be included in a language, a commonly agreed set of evaluation criteria is needed. These criteria will by definition be controversial, as whether a programming language is efficient to read and write programs in, is object to subjective opinions of the developer. This being said, the evaluation criteria describe what we focused on in our programming language. This section begins with a brief description of the different evaluation criteria. The content of this section is based on [22].

2.10.1 Readability

Readability focuses on how easy a written program is to read and understand. A programming language should contain a small number of basic constructs. This requires less of the reader hereby enhancing readability. A large amount of basic constructs can also lead to feature multiplicity, that is having more than one way to perform a particular operation. In Java, for example, the readability is affected negatively by having multiple integer increments:

Listing 2.8: Multiple ways off increasing a variable in Java

```
count = count + 1;
count += 1;
count++;
++count;
```

All the statements in listing 2.8 above increments the variable `count`, when used seperately.

2.10.2 Orthogonality

Another important aspect to consider in a programming language is orthogonality. Orthogonality is achieved when a small set of primitive constructs can be combined in a limited way to form the language's control and data structures. For example, in listing 2.8 orthogonality is not very high because of the many ways a variable can be incremented. If orthogonality is achieved, or is high, in a language there will be fewer exceptions to its rules. This leads to a more regular language that is easier to learn and understand.

2.10.3 Writability

Most of the characteristics that define the readability and orthogonality of a language also has an impact on writability. Writability is defined by how easy it is for the programmer to write a program for a specific problem domain. The problem domain is very important when considering writability because of the problems nature. For example, a language for mathematicians for writing computational algorithms should have a high writability, whereas a language for modelling molecules might focus more on readability. In most cases writability is at odds with readability because if simplicity is achieved in writability, the code will be harder to read. Likewise if simplicity is achieved in readability then it will be a more strenuous task to write programs in for a developer that knows the language.

Another aspect that has a huge impact on writability is abstraction. Abstraction is the ability to reuse structures or operations that have been previously defined. Functions is a great example of abstraction, because they only need to be defined once and can then be used an unlimited number of times by invoking a function call.

2.10.4 Reliability

Reliability is achieved in a programming language if the language processor includes features that ensures that the program performs as specified in its definition. Reliability is interconnected with readability, because a better understanding of the program will result in catching more errors.

Type checking is the most used feature for ensuring the reliability of a program. When processing the language the processor checks if data types are used in agreement with the language definition. This helps ensure that the program does not crash at run-time due to faulty use of data types.

A language feature that is able to intercept run-time errors is *exception handling*. This feature enables the developer to write constructs in the program that catches possible errors, handles them in a given way and continues the execution of the program without crashing.

A scenario that has an impact on reliability is *aliasing*. Aliasing is when two variables point to the same address in memory, i.e. the same data. This can create a problem if one of the aliased names changes the data unexpectedly without the developer's knowledge.

2.10.5 Other Criteria

Besides the four criteria already mentioned, there are numerous other language criteria that could be considered when designing a programming language. The list below briefly describes a selected few of these other criteria.

- Maintainability: Concerns ease of adding new features and finding errors (debugging etc.).
- Extensibility: Adds features that allow the user to add new constructs to the language (user-defined types etc.).
- Consistency: Consistency with regular conventions and notations, which makes it easy to learn by experienced programmers.

Even though we might implicitly include some of the many other language criteria when designing the language for this project, we delimitate the choice by only focusing on the first four mentioned in the subsections above.

2.10.6 Choice of Language Criteria

The design target criteria for the language is readability as our main priority, writability as second and reliability as well as orthogonality as third. The following table illustrates how we prioritize the various criteria in our language.

	Level of importance			
Language Criteria	**Very**	**Somewhat**	**Less**	**None**
Readability	X			
Writability		X		
Orthogonality			X	
Reliability			X	

Table 2.1: Language criteria importance

Readability has high priority, since our language is supposed to be simple and easy to read.

Writability has somewhat high priority, as our language is targeted at problems within a domain that would gain from being translated into parallel code. Moreover, we wished to include functions in our language, which also enhance writability, by means of abstraction.

Orthogonality is chosen because having a small set of primitive constructs suited the problem domain best, as it would ease the dependence analysis. Furthermore, having a complex language with many different constructs is not the purpose of this project.

Reliability is prioritized in our language as well, because correct use of data types is important when translating it to Java, even though Java includes run-time type checking.

To summarize the language for this project should be easy to read, fairly easy to write, include a small set of basic constructs and have a reliable type system.

2.11 Problem Statement

This section contains the problem statements, which is derived from the analysis done until now and narrows down the initiating problem. That is, this problem describes the focus for the rest of this report.

- How do we develop a programming language, that can be translated into parallel code, with the use of a translator?
    - How do we define the syntax of such a programming language?
    - How do we define the semantics of such a programming language?
    - How do we achieve the language criteria defined in section 2.10?
- How do we develop a translator, that automatically generates parallel code from sequential code?
    - How can we tokenize source code?
    - How can we parse source code?
    - How can we check source code for contextual errors?
    - Should the parts of the compiler, that can be generated by tools, be generated, or should the whole compiler be handmade?
    - How do we analyze source code written in the programming language, and determine which parts can be parallelized?
    - How do we generate parallel code?

We want to create an imperative, implicit parallel language, which is simple and relatively easy to learn. For this language, a compiler, that is able to meet our requirements, will be made as well. For future references, we will name our upcoming programming language Imp.

2.11.1 Problem Delimitation

Given the limited time for making this project, we will focus on having a working translator including all the compilation phases, making loops concurrent if possible, which will give some proof of implicit parallelism, and fulfill the required criteria we have set in the analysis. However, our translator will only produce Java files and we are not able to implement a fully implicit parallel language. Ideally, we want to achieve implicit parallelism for our entire language, but this will not be possible, due to our time limitation.

2.12 Summary

Throughout this analysis, various topics have been covered to get a better understanding of how to develop a translator, that includes implicit parallelism. In section 2.1 we compared two types of language processors, translators and interpreters, and chose to build a translator because it is more suitable for the focus of this project. Analyzing the various compilation phases in section 2.2 gave us a better understanding of the steps we need in our compiler in order to make sequential code parallel. In sections 2.4 and 2.5 we studied parallel computing to be able to distinguish between various forms of concurrency and approaches to implicit parallelism. In section 2.6 we analyzed topics important for dependence analysis to be able to define how much data analysis should be done by our translator. Sections 2.9 and 2.10 defined our development and target language. Finally, the Problem Statement in section 2.11 specified our problem domain and delimited the focus of this project.

Chapter 3
Design

In order to make a programming language from scratch, we need to make a lot of design decisions before we can start implementing. In the following chapter, we will draw use of the theory that we discussed in the analysis and determine how we potentially can use it in our implementation of the Imp translator. This chapter contains design discussions about the different elements of our upcoming programming language. Furthermore, the chapter will contain design decisions such as how we are going to analyze the source code for potential concurrency (section 3.8), how we are going to use it in our code generation as well as how we are going to design the compiler in general (section 3.2).

3.1 Language Specification

This section specifies the Imp language in terms of its syntax, type system, control structures and semantic in the form of an informal semantic. Moreover, we elaborate on how types in Imp relate to types in Java, and also which features should be included in the standard library.

3.1.1 Syntax

The first step in defining the Imp programming language will be to define its syntax. As mentioned in the syntax part of the analysis (2.2.1), this is done by defining an EBNF for our language. As described in the language criteria section in the analysis (2.10), we are going to focus on readability. We have been inspired by the syntactical elements from C and Java and we plan to base the Imp language on these two languages. By letting the syntax be similar to other mainstream programming languages, readability can be achieved. It also makes the language easier to learn hereby enabling developers to quickly get started utilizing implicit parallelism.

The first step of defining the CFG in EBNF is to agree on the phrase structure of our compiler. How do we want our syntactical appearance of our programming language to be like? To answer this, we have made an example of a piece of code that we would like to see as being syntactically valid once our language has been developed. The code can be found in appendix D.5 and the CFG of the language can be found in appendix A. An excerpt from our EBNF, the `Single-command` production rule, is displayed below in listing 3.1.

Listing 3.1: EBNF for `Single-command`

```
Single-command  ::= FName ( Function-call-parameterlist );
    |   Name = (( FName ( Function-call-parameterlist ) |
        Expression ));
    |   [expression]Name = (( FName ( Function-call-
        parameterlist ) | Expression ));
    |   if ( Bool-expression ) { Command } [[else { Command }]]
    |   while ( Bool-expression ) { Command }
    |   foreach (Type-denoter Name in Name) { Command }
    |   { Command }
    |   Declaration
    |   Returner ;
```

We want our language to be LL(1) due to the fact that we plan to write a recursive descent parser. There are several other parsing algorithms used in compilers, e.g. to implement the CFG in a parsing table, but as "Programming Languages Processors in Java" [24] states, a RD parser "is both effective and easy to understand". Since this is a project where learning is of great importance, this algorithm has been chosen. This means, that we have produced our EBNF so that no ambiguity occurs, thereby satisfying the conditions that the classification of LL(1) languages have.

3.1.2 Type System

We have selected our types to be very similar to some of the ones in both C and Java. Basing our language on one that already exists will enhance readability, at least for programmers accustomed to C. Our five primitive data types are listed below.

- int. Defines integer numbers e.g. int number = 42
- float. Defines floating point numbers. That is two integers separated by a period e.g. float number = 3.14159
- char. Defines characters e.g. char letter = 'F'
- string. Defines a character string e.g. string words = "Here is a string"
- void. Defines the absence of a type e.g. a function that does not return anything, must have the return type void.

Besides the primitive types, we also want to implement the data structure, array. Our syntax for arrays will be similar to the one in C using square brackets, however our array will be a simpler than the C array. We will only be able to define an array with a certain length, insert to and extract data from a certain cell. Compared to C our language shares the simple types: int, float and char. The differences are that our language has strings, while C has different extended number types such as double and long int. Like the C programming language our language does not contain a boolean type. Below is an example of how arrays work in Imp.

- array char [1]charArray;

0 charArray = ’c’;

Compared to Java, Imp only has a few data types and our string is not as useful as the Java string, because the Java string is a special class included in the `java.lang` package and therefore actually not a primitive data type. Java also has other number types like long and double. However, our choice of types is not arbitrary. We see it as a big advantage that all the types we have also exists in Java, this also makes it easier to handle types when the translator generates the Java code.

3.1.3 Operators

Computing expressions is a very important part of a programming language. To do this, operators are needed. This section contains a description of the operators included in Imp, being arithmetic and boolean, and compares them to some of the similar operators in Java and C. All the operators of Imp are included in both Java and C. Generally in Imp, expressions are evaluated using left to right association, so the expression `a + b + c` corresponds to `( a + b ) + c`. The content in this chapter is based on [22].

Arithmetic Operators

Arithmetic operators are for specifying an arithmetic computation. In Imp, we only have binary operators that are infix, which means an operator must have two operands with the operator in between in contrast to unary, that only have a single operand. Operators that are prefix precede their operands. Table 3.1 below shows the arithmetic operators in Imp compared to the equals in Java and C.

Type	**Imp Syntax**
Add	a + b
Subtract	a - b
Multiply	a * b
Divide	a / b
Parenthesis	(a)

Table 3.1: Arithmetic Operators in Imp

Parenthesis is not actually an arithmetic operator, as it cannot express an arithmetic computation, but the parenthesis is important for the precedence of the calculations. Precedence rules for arithmetic calculations in Imp are given in its syntax (see appendix A), where parenthesis, multiply and divide has precedence over add and subtract.

Both Java and C allow increments of values with unary operators that are prefix (`++a`) and the modulus (`a % b`). We have decided not to include these in Imp, due to our language criteria, where readability and simplicity is prioritized. We do not see these operators as self explanatory and therefore not in compliance with our language criteria.

Relational Operators

A relational operator is an operator that compares the values of two operands. Imp includes some of the relational operators in C and Java. Table 3.2 contains a list of the operators available in Imp.

Type	**Imp Syntax**
Equal	a == b
Not equal	a != b
Greater than	a > b
Smaller than	a < b
Greater than or equal	a >= b
Smaller than or equal	a <= b

Table 3.2: Relational Operators in Imp

Logical Operators

The logical operator `&&` takes two boolean expressions and is true if both expressions are true, the || operator also takes two boolean expressions and returns true if either one is true. Both operators are short-circuited which means that the `&&` does not calculate the second expression if the first is false, likewise for || if the first expression is true. The logical && operator have higher precedence than the || operator, which means that in a boolean expression the and operators are evaluated first.

Type	**Imp Syntax**
Or	a || b
And	a && b

Table 3.3: Logical Operators in Imp

3.1.4 Assignments

Assigning a value to a variable is of great significance in an imperative language like Imp. It also gives the possibility to change the value of a variable dynamically. This section describes how globals, constants, variables and arrays are assigned. In general, declarations are only legal when initialization is carried out. That is, in Imp you have to assign a value to a declaration. Moreover, types in Imp are strongly bound to the variable, so types cannot be changed once the variable is declared.[22]

Globals

A variable declared outside the `Main()` function with the keyword `global` preceding it, is considered a global variable. Listing 3.2 gives an example of this.

Listing 3.2: Global variable declaration

```
global float testGlobal = 1.23;
void Main () {
    ...
}
```

The global variable can be used in any scope and can be change anywhere in the program. In C the syntax is the same as ours without the `global` keyword, and as Java is purely object oriented there is no global variables outside the classes.

Constants

A constant is declared with the keyword `const` and its name following it. Listing 3.3 is an example of a constant declaration.

Listing 3.3: Constant declaration

```
void Main () {
const int testConst = 80;
}
```

A constant cannot be assigned to other variables, only numbers, strings or symbols. It cannot be changed and can only be used in its own scope and its child scopes. The syntax is identical to C. Java lacks a `const` keyword though it has a `final` keyword with similar functionality.

Variables

A variable is assigned to a value with the assignment operator. Furthermore, variables can be assigned to other variables if their types match. Below is an example that shows these incidents.

Listing 3.4: Variable declaration

```
void Main() {
    int a = F();
    int b = a;
}
int F() {
    int c = 2+1;
    return c;
}
```

Listing 3.4 above shows how functions can be assigned to variables as well. The variable `c` ends up being equal to the value 3. In Java and C it is possible to use a binary operator followed by the `=` operator. For example: `a += b`, is equal to `a = a + b`. Since Imp is supposed to have high , we have not included this option.

Arrays

All types can be created as arrays by using the keyword `array` in front of the type succeeded its variable name and a number or expression to denote the array length in between a "[" and a "]". Listing 3.5 shows how it can be done.

Listing 3.5: Array declaration

```
void Main() {
    array int [3]testArray;
    [0]testArray = 1;
    [1]testArray = 2;
    [2]testArray = 3;
}
```

Arrays in Imp are static, which means the length of the array cannot be changed. The syntax of arrays in Imp is similar to C and Java, the differences are the declaration which requires the `array` keyword and the "`[n]`" block in front of the variable name, in contrast to C and Java where it is after. This is because our grammar is LL(1) and if the brackets were after the array name, the parser would confuse them with regular variables. Likewise, for accessing array cells where the square brackets are in front of the array name.

3.1.5 Control Structures

This section describes the design of our control structures in Imp, being conditional branches and loops.

Conditional Branches

As written in section 2.10, we want a fairly high orthogonality. For that reason we have chosen only one way to write the conditional command, the `if-else`.

Listing 3.6: Example of our `if-else` construct. The else part is optional

```
if (boolean expression) {
    ifTrueCommands
} else {
    ifFalseCommands
}
```

Our `if-else` construct does not support `'else if'` that are typically seen in other languages, instead one have to use nested `if-else` commands.

Loops

In Imp we have two different loop constructs: The traditional `while` loop and the `foreach` construct. We have included the `while` loop because all imaginable loops can be expressed with this construct. We have also included the `foreach` loop because it is easier to analyze regarding dependencies.

The `while` loop is the general purpose loop that can iterate over any kind of data, while the `foreach` only iterates over arrays, and each iteration corresponds to a single cell in the array.

Listing 3.7: Calculating the Fibonacci sequence

```
int previous1 = 0;
int previous2 = 1;

while (previous2 < 100) {
    int temp = previous1;
    previous1 = previous2;

    previous2 = temp + previous1;
    Printi(previous2);
}
```

Listing 3.7 above shows how a `while` loop can be created in Imp to calculate the Fibonacci sequence. As illustrated, the syntax of the `while` loop is the same as in Java and C. To keep the language simple we have neither `do-while` nor `for` loops in our language since they do not add any new functionality to the language that it does not already include. Any `do-while` or `for` loop can replaced with a `while` loop.

Listing 3.8: Calculating the sum of numbers in an array

```
...
float sum = 0;
foreach(float f in numArray) {
    sum = sum + f;
}
...
```

The syntax of the `foreach` loop (see listing 3.8) is inspired by the one in C# , and is fairly straightforward. In listing 3.8 the `cell` variable is declared locally and points to a single cell in the `numbers` array, that are defined outside the code listing.

3.1.6 Functions and Procedures

To create readable and maintainable code, it is our belief that functions are useful. If we did not have functions one may have to copy/paste code around the program, which is error prone because if there is a bug in the code that is copied, the user would have to correct the bug multiple places to fix it. The user can avoid this if the code is separated into functions. As stated in setion 2.10, another concern is readability as the user can put complex code into a well named function.

While some languages have more than one way to create a function, Imp only have one way to create functions. This way we keep the syntax clean and simple.

Listing 3.9: Calculating Fibonacci sequence

```
void Main() {
    Print("Calculating Fibonacci");
    int previous1 = 0;
    int previous2 = 1;

    while (previous2 < 100) {
        int temp = previous1;
        previous1 = previous2;

        previous2 = Add(temp, previous1);
    }
}

int Add(int x, int y) {
    int temp = x + y;
    return temp;
}
```

Listing 3.9 calculates the start of the Fibonacci sequence, but this time we have separated the addition logic from the while loop into a function, `Add`, that returns the result. First there is the `Main()` function that have the special type `void`. The `Print()` function is part of the standard library for Imp, discussed more in section 3.1.7. The `Main()` function is the entry point of a program and can take an arbitrary number of parameters, which are loaded from the command line when the program is executed.
The syntax should be quite familiar to anyone who has used a C-derived language before. The syntax will be the same, except for the naming convention for functions, where they must be started with a capital letter instead of lowercase.

3.1.7 Standard Library

We have decided that libraries cannot be included in Imp source code, unlike most major programming languages. Our reason for this limitation is that we have no need for the kind of advanced programs that one would need to include libraries for - we only need to write programs that demonstrate simple concepts of implicit parallelism and programming language development in general.

Instead, we have decided that certain functions that might be used in programs commonly, can be called without prior declarations, for instance `length(someArray)`, which returns the number of elements in an array, or `sqrt(float)`, which returns the square root of a floating point number. Here is a list of the functions in the Imp standard library.

- Print Functions
  - Print(string) : void
  - Printi(int) : void
  - Printf(float) : void
  - Printc(char) : void

- Array Functions
  - Lengths(string array) : int
  - Lengthi(int array) : int
  - Lengthf(float array) : int
  - Lengthc(char array) : int
- Math Functions
  - Cos(float) : float
  - Sin(float) : float
  - Sqrt(float) : float
  - Sqrti(int) : float
  - Mod(int, int) : int
  - Pi() : float

It is not possible to write any logic in Imp that prints output. This is why we have added print functions to the standard library.

3.1.8 Semantics

It is important to have a precise and expressive semantic for a programming language prior to implementing it, because ambiguity and vagueness in the semantic may lead to different implementations. In this section the type of semantic used in this project is choosen and then the first part of the semantic itself is elaborated. The rest of the semantic is located in appendix C[12].

Choosing Semantics

First we will take a look at the formal semantics. Axiomatic semantics are not suitable for this project because it focus on why a program behave in a certain way and is more beneficial if correctness is a high priority in the language design. Using denotational semantics would make it hard to express parallelism and therefore we do not choose this standard neither. Because operational semantics (OS) focuses on how a program is executed, it would provide a solid basis for defining parallelism in Imp. Due to Imp having parallelism, using big-step will not be an option. The fact that big-step semantics model a transition in one step makes it impossible to know all the states a variable might be in. Small-step semantics on the other hand defines a transition in a series of steps, hereby allowing us to make dependence analysis on variables based on their state. Even though small-step semantics might be a good way to define a language containing parallelism, we will not use it in this project. This is because making a complete semantic using small-step would move the focus of the project away from developing an implicit programming language, and into a more theoretical approach. Another reason is that the Syntax and Semantics course focused on big-step semantics, leaving a lot of uncovered material on small-step semantics left for us to study outside the curriculum of the Syntax and Semantics course, especially semantics for describing parallelism.

We will define our language by creating an informal semantic. This decision is based on keeping the focus of the project on actually developing the translator and not just define the language. While still getting a semantic that gives a complete blueprint for implementing the Imp programming language. Now that we have chosen the type of our semantic, we will move on to actually making one for the Imp language.

Informal Semantics For Imp

The following pages define the semantics for the Imp programming language by describing each of the production rules from our EBNF one at a time. We start with the most abstract rules like `Program` and `Function`, then move on to `Command` and `Expression`, ending at `Literal`. This order is similar to the structure of an abstract syntax tree, starting at the root node (`Program`) and end in atomic child nodes (`Literals`).

```
Program ::= (( global Declaration ))* Function
```

`Program` is the root production rule for the Imp language. It defines that a program written in Imp may have none or many `Declarations` with the keyword `global` in front. The `global` keyword ensures that any variable declarations that are outside a function needs to be placed in the beginning of the program prior to any function declarations.

- `Program` is executed as follows: First, any possible global `Declarations` are evaluated and executed; then the `Function` is executed.

```
Declaration ::= Type−denoter Name =(( String−literal | Expression )) ;
| const Type−denoter Name = Literal ;
| array Type−denoter Name[ Expression ] ;
```

This production rule defines declarations of variables, constants and arrays. All declarations are ended with a semi-colon and in all cases the `Type-denoter` type is bound to the `Name` identifier.

- The declaration ”`Type-denoter Name =(( String-literal | Expression ))`” is elaborated as follows: First, the `Type-denoter` is evaluated to yield a type; then `Name` is evaluated as variable identifier; then the `Name` variable is updated with either a `String-literal`, evaluated to a string value, or an `Expression` evaluated to a value. In both cases the types of `Name` and the `String-literal` or `Expression` must be equivalent.

- A constant declaration is elaborated as follows: The constant is denoted by a `const` keyword, thereafter its `Type-denoter` is evaluated to a type; then `Name` is evaluated as variable identifier; the `Name` identifier is updated with the value of the evaluated `Literal`. The types of the `Name` and `Literal` must be equivalent.

- The last form of declaration is an array declaration. Such a declaration is started with an `array` keyword, a `Type-denoter`, the name of the array (`Name`) followed by two square brackets containing an `Expression`. The square brackets contain the size of the array and can be the result of an `Expression`. The array declaration is elaborated as follows: First the array is denoted by an `array` keyword, whereafter the `Type-denoter` is evaluated to a type; then `Name` is evaluated as variable identifier; then the `Expression` is evaluated to a value. `Expression` type has to be an integer.

```
Function ::= (( Function−declaration ))+
```

The `Function` used in `Program` consists of one or more `Function-declarations`. This rule dictates that an Imp program must have at least one function declaration and can have many consecutive ones.

- `Function` is elaborated as follows: If several function declarations exists, F_1; F_2, the F_1 is elaborated first, then F_2 is elaborated.

```
Function−declaration ::= Type−denoter FName ( Parameterlist ) {
    Command }
```

A function declaration consists of a type, a function name (`FName`), a `Parameterlist` enclosed by parenthesis and a `Command` enclosed by square brackets. The type of a function is the type of the value it may return (the type can also be `void`). If the type is `void` then no return statement needs to be included in the function declaration. The `Parameterlist` contains any parameters the function might be supplied with when invoked. The `Command` contains the entire body of the function.

- `Function-declaration` is elaborated as follows: `Type-denoter` is evaluated to yield a type, `Parameterlist` is evaluated to yield a list of parameters, elaborated below, `Command` is evaluated to yield either one or more `Single-commands`, also elaborated below. The `Type-denoter` type and the list of parameters is bound to the `FName` identifier.

```
Parameterlist ::= [[ Parameter ]]
```

- A `Parameterlist` is associated with either zero or one `Parameter`, elaborated below. This enables function declarations to have either zero or more parameters defined.

```
Parameter ::= [[array]] Type−denoter Name
| [[array]] Type−denoter Name , Parameter
```

A `Parameter` is either a single parameter or a parameter followed by a comma and another `Parameter`. This enables a `Parameterlist` to contain either one or many `Parameters`. A parameter is denoted by a type and a name with the possibility to add the `array` keyword in front of it to allow passing arrays as parameters.

- The parameter "`[[array]] Type-denoter Name`" is evaluated as follows. First either zero or one `array` keyword is checked; thereafter the `Type-denoter` is evaluated to a type; then `Name` is evaluated as variable identifier. The `Type-denoter` type is bound to the `Name` identifier. If an `array` keyword exists, the `Name` is an array.

- The sequential parameter is executed as the item just above, but evaluates the `Parameter` once again to gain multiple parameters.

```
Function-call-parameter ::= Name , Function-call-parameter
| Literal , Function-call-parameter
| Name
| Literal
```

Parameters used in function calls differ from the parameters used in function declarations. `Function-call-parameter`s are either a `Name` or `Literal` possibly followed by another `Function-call-parameter`. A `Name` denotes a variable name making it possible to pass variables as arguments to function calls. A `Literal` denotes a value of one of our four simple data types, which enables typing values directly into the function call. `Literal`s and `Name`s are separated by a comma. `Function-call-parameter` is executed as follows.

- Sequential function call parameters are evaluated one at a time. First `Name` is evaluated to yield a variable identifier then the next parameter is evaluated.
- First `Literal` is evaluated to yield a value, then the next parameter is evaluated.
- `Name` is evaluated to yield a variable identifier. The type of `Name` must correspond to the type declared in the function declaration.
- `Literal` is evaluated to yield a value. The type of `Literal` must correspond to the type declared in the function declaration.

```
Function-call-parameterlist ::= [[ Function-call-parameter ]]
```

A `Function-call-parameterlist` contains either zero or one `Function-call-parameter`. This enables function calls to have either zero or more parameters defined.

- `Function-call-parameterlist` is associated to a list of function call parameters.

```
Command ::= Single-command
| Single-command Command
```

A `Command` contains either a `Single-command` or a `Single-command` followed by a `Command`. This production rule makes it possible to have many sequential `Single-command`s where a `Command` is located, for example in a function declaration.

- The command `Single-command` is executed as follows. The `Single-command` is evaluated to a command.
- The sequential command is executed as the item just above, but evaluates the `Command` once again to get multiple commands.

```
Single-command  ::= FName ( Function-call-parameterlist );
| Name = (( FName ( Function-call-parameterlist ) | Expression ));
| [Expression]Name = (( FName ( Function-call-parameterlist ) |
   Expression ));
| if ( Bool-expression ) { Command } [[else { Command }]]
| while ( Bool-expression ) { Command }
| foreach (Type-denoter Name in Name) { Command }
| { Command }
| Declaration
| Returner ;
```

The `Single-command` production rule contains most of the functionality of the Imp programming language. In the following, we elaborate on the nine different sub-rules.

- The "`FName ( Function-call-parameterlist )`" sub-rule defines function calls. A function call is characterized by a function name (`FName`) and a parameter list (`Function-call-parameterlist`) enclosed by parentheses and ended with a semicolon. The parameter list can also be empty if the corresponding function declaration parameterlist also is empty.
  It is executed as follows. The `FName` is evaluated as a function identifier; then the `Function-call-parameterlist` is evaluated to output an argument list. The `FName` must be bound to a function and the parameters must be compatible with this `FName` function.

- The "`Name = (( FName ( Function-call-parameterlist ) | Expression ));"` assigns the result of function calls or expressions to a variable. It is executed as follows. The `Name` is evaluated as variable identifier; thereafter either a function call or an expression is evaluated.

  – The `FName` is evaluated as a function identifier with bound parameters, as elaborated in the item above, and the return value of the function updates the `Name` identifier with this value.

  – The `Expression` is evaluated to a value; then the identifier `Name` is updated with this value.

  `Name` must be bound to a declaration. The types of `Name` and return type of `FName` or `Expression` must be equivalent.

- The next sub-rule, the array sub-rule, is equal to the sub-rule in the item just above, except the assignment is stored in an array instead of a `Name` identifier. The square brackets contain the index of the array, where the values is to be stored. The array assignment is executed as follows. The `Name` identifier is checked to see if it is an array. If `Name` is an array, the `Expression` is evaluated. The `Name` array identifier is evaluated and updated with the value of either the function call or the expression, explained in the item above. The `[[Expression]Name` must be bound to an array declaration. The `Expression` must be an integer type.

- The "`if ( Bool-expression ) { Command } [[else { Command }]]`" single command starts with the keyword `if` and a `Bool-expression` enclosed by two parenthesis. The command executes as follows. The `Bool-expression` is evaluated, if its value is "1", then $Command_1$ is executed. If the `Bool-expression` value is "0", then *Command*$_2$ is executed. The `else` statement is optional, meaning that an `if` does not require a corresponding `else` to be syntactically correct.

- A while loop is initiated by the keyword `while` and is executed as follows. A `Bool-expression` is evaluated, if its value is "1", then *Command* is executed. The while-command is looped. If the `Bool-expression` is "0", the while-command is stopped.

- The `foreach` loop iterates through the contents of an array.
  The "`foreach (Type-denoter` N_1 `in` N_2`) Command` " command is executed as follows. The `foreach` keyword is evaluated. Then the `Type-denoter` is evaluated to a type; N_1 is evaluated as variable identifier. The `in` keyword is evaluated and the N_2 is evaluated as an array identifier. `Command` is executed.
  For each item in the array, N_2, the current field in the array is stored in a variable, N_1, that is accessible in the `Command` inside the square brackets of the loop. The `Type-denoter` of N_1 has to correspond to the type of the array (N_2). `foreach` loops are potentially parallel, meaning that if they uphold the dependence rules for Imp, the iterations will be split into separate executable threads. Checking if a `foreach` loop is parallelizable is done through loop dependence analysis (covered in section 3.8).

- The "{ `Command` }" command is executed as follows. The `Command` is executed, as elaborated above.

- " `Declaration` " is a used to declare variables, constants and arrays. The `Declaration` is executed, as elaborated above.

- `Returner` is used to return a value in a function declaration. The `Returner` is executed.

The rest of the semantics for Imp are located in appendix C.

3.2 General Translator Design

Before designing a translator there are some decisions that have to be made. That is, it is important to decide how the translator is meant to work. For example, does it include all the phases in one or two steps? Is it to be generated or made by hand?

As mentioned in the introduction of the report and the problem statement (section 2.11) the idea of this project is to create a programming language that can utilize concurrency and potentially take advantage of multi-core processors. Hence, we aim to move the responsibility of concurrency from the programmer to the translator. This will allow the programmer to write programs without worrying about which processes can run concurrently. It is the task of the translator to analyze the written program and decide if and how the program can be split into multiple execution paths that are able to run independently. Since Java is used as target language (see section 2.9), threads can be created to execute these blocks of code (more on this in section 3.9).

A solution would involve translating a program written in our language, Imp, into a Java program. During translation, the program will be analyzed for dependencies and this will provide the rules for splitting the program into multiple threads. This has to be done without giving any special instructions or changing the program. Moreover, the translated program should still have the same input and output as the developer intended.

We have two different approaches to the overall design of the translator. The main difference between the two, is that one uses a single language and the other uses two. To show programs and translators we will use *Tombstone Diagrams*. Tombstone diagrams visualizes different components, e.g. machines, compilers and languages, and how they function in relation to each other[24]. It is important to notice that adjacent sides of the diagrams indicate the same language.

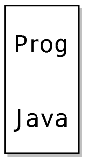


Figure 3.1: This represents a program written in Java[24].

To describe tombstone diagrams further, some simple diagrams are showed in figure 3.1 and 3.2. Figure 3.1 illustrates a program written in Java. Figure 3.2 illustrates a Java program that is translated to x86 by means of a x86 machine.

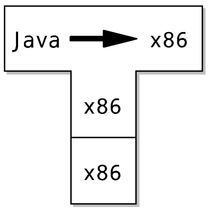


Figure 3.2: This represent a translator with Java as source language and x86 machine code as target language. The translator itself is also written in x86 and runs on an x86 machine. Ref.: [24]

First approach will include a design of two languages: `lang1` and `lang2`. These languages will be equal except that `lang2` will have syntax for defining which blocks of code are able to run in parallel. The approach consists of two translators, see figure 3.3, done in two steps:

1. The first step will translate `lang1` to `lang2`.
   This step includes all compilation phases as well as the dependence analysis, where code is explicitly defined as concurrent parts.

2. The second step will translate the the explicit parallel `lang2` into Java, which uses threads.

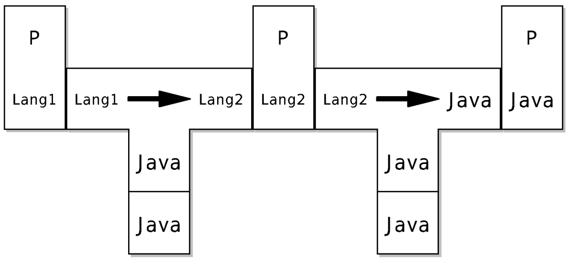


Figure 3.3: Translator design, first approach.

Second approach will include translating a language, `lang1`, directly to Java, as illustrated by figure 3.4, running the compilation phases and dependence analysis before the code generation. This only requires one translator whereas the first approach requires two very similar translators. After the dependence analysis, this approach will generate a Java output, contrary to the first approach which generates a `lang2` output. The advantage of the second approach is the option to see or even write the program directly in `lang1`, before translating it into Java.

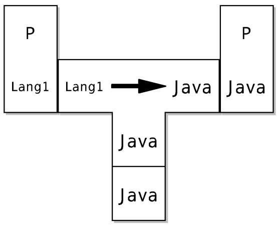


Figure 3.4: Translator design, second approach.

The construction of two translators, as proposed in the first approach, will be similar, yet still have some differences that could make development a slightly trivial and bigger task. Thus, we have chosen to use the second approach, because we think it will be unnecessary to write to an extra language, `lang2`, in the first approach, when the dependence analysis already has been taken care of. To be exact, when the dependence analysis is carried out directly on the AST (see more in section 3.4), we do not see a big advantage in first outputting `lang2` and later Java. The second approach enables us to focus on parallelism while still including all the other steps of compilation.

The implementation of the translator will be made by hand, since we find it valuable to learn and understand how to build a translator, besides, we find it important to know exactly what the different parts of the compilation phases do and we hope that all in all we will learn more about programming languages. Figure 3.5 shows an overview of our translator design, which consists of the following phases written about further ahead:

- Syntactical analysis, sections 3.3, 3.4 and 3.5.
- Contextual analysis, section 3.6.
- Loop dependence analysis. section 3.8.

- Code generation. 3.9.

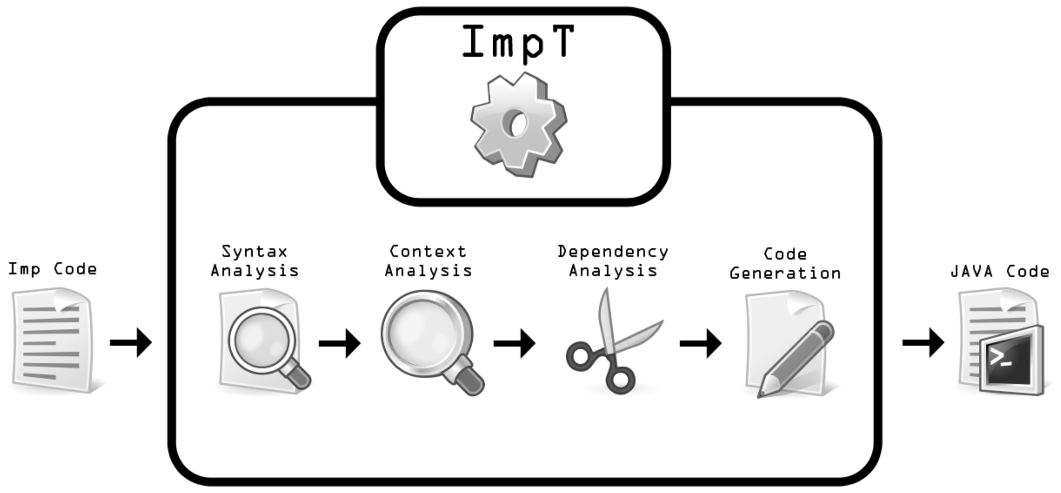


Figure 3.5: Overview of the translator.

The following sections will describe how these compilation phases will be designed.

3.3 Tokenizer

In order to construct a tokenizer, we need to use the theory obtained from section 2.2.1. This section contains a description of how we plan to construct our tokenizer and what information our tokens will contain.

The purpose of a tokenizer is to divide our source code into tokens so we can use it as a part of our abstract syntax tree. In order to obtain these tokens, we need a way to identify and retrieve them. The way we plan to do this, is by using Java's regular expressions. The reason why we want to use these in our implementation, is because it is a strong tool provided by Java, which enables us both to identify and retrieve the tokens from the source code. It would be possible to use other methods for the tokenizer, such as reading the entire source as a long string and checking every word of the string by using a large switch statement. This however, can be avoided using regular expressions. This might give us more flexibility in terms of the acceptance of words, because we can more easily edit the regular expressions instead of editing a large switch statement.

To satisfy future requirements of the product development, we want the tokens to contain information about what kind of token they are and where in the source code they are located. The reason why we want the tokens to be named according to their kind, is because we want to use them in the construction of our abstract syntax tree.

We want the tokens to have a line number, because we want our product to be able to identify the line where an error occurs so the user of Imp will be able to find the error and correct it. Therefore we want the tokenizer to be able to handle line numbers at an early stage of development. The more important information we can attach to the tokens themselves, the less we need to worry about tracking down the origin of tokens at a later stage of the Imp translator.

3.4 Abstract Syntax Tree

Since we are making a multi-pass translator, we need some form of representation of the source program to share information in between the various steps of the translator. Such a representation could be a tree or a table. The problem with using a table is that it does not give an intuitive representation of the structure of the program. A tree representation on the other hand could provide this. In the following we describe how such a tree could be designed.

As explained in section 2.2.1 each node in the tree should correspond to a production rule in the CFG. Because we are using an object-oriented programming language (Java), a way to implement this would be by creating a class for each type of node. Each of these classes should contain attributes corresponding to the child and parent nodes the production rule defines. All the classes should also contain a visitor method to enable the use of the visitor design pattern to traverse the tree (see more about this in section 3.10). For example, the class representing the `Program` production rule should contain the following illustrated in figure 3.6.

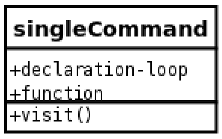


Figure 3.6: Class diagram for the `Program` production rule.

3.5 Parser

When constructing a parser, it is appropriate to look at the source program as tokens, as described in section 2.2.1. These tokens are used by the parser to build up the phrase structure in an AST. The task of the parser is to check if an input string is part of the grammar and then represent it as an AST, if this is the case. When a machine is to parse a grammar, the grammar needs to be unambiguous, which means that every sentence of the grammar only has one syntax tree.

As mentioned in section 2.2.1, there are two strategies for parsing: bottom-up and top-down. These are named by the order of which the AST is constructed. Since we have chosen to write a recursive descent (RD) parser (see section 3.1.1), this section is based on [24] and [22], and will describe the RD algorithm and how the parser is meant to work.

The RD algorithm uses the top-down parsing strategy. The RD parser consists of parsing functions for each production rule in the grammar. Many of these functions are recursive, hence the name of the algorithm, since statements are often nested in other statements. The task of each function is to parse a single node in the AST. All the parse functions cooperate to parse the complete AST in the end.

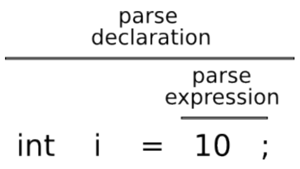


Figure 3.7: Recursive Descent Parsing of an Imp Sentence.

3.6 Contextual Analysis

As mentioned in section 2.2.2, the purpose of the contextual analysis is to make sure that the input program complies with the contextual constraints defined for the language. Contextual constraints consist of scope rules and type rules. However, if the language is dynamically typed, the constraints will not contain type rules. The source for this section is [24].

Scope rules regulate applied occurrences of identifiers and declarations. The left part of figure 3.8 shows the binding occurrence of `pi` and `realPi` and their applied occurrences. Binding occurrence is when a variable is initialized. Type rules provide information about the types of expressions and decide whether these expressions have valid types compared to their declaration or if they have been declared it all. The right part of figure 3.8 shows an incident where the variable `y` has not been bound, which will result in an error message.

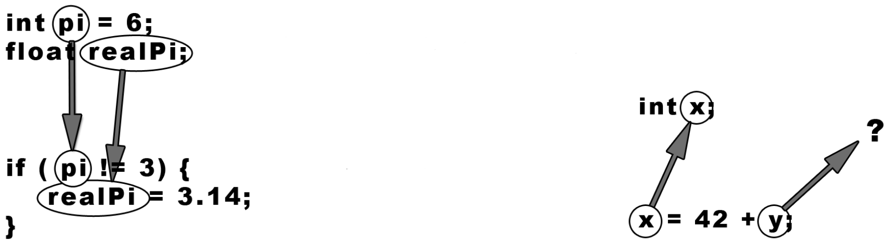


Figure 3.8: Contextual constraints

Contextual analysis consists of two sub-phases: Identification and type checking. In the identification sub-phase the scope rules are applied, so that each applied occurrence of identifiers are related to their declarations. In the type checking sub-phase, the type rules are applied to let the type checker know the types of each of the expressions and compare them with the expected (declared) types. A more in depth explanation is written in the following subsections.

3.6.1 Identification

In the identification sub-phase, we relate applied occurrences of identifiers to their corresponding declarations, and report errors if identifiers are used in incorrect ways.

Since the source program is represented by an AST, a possible way to do the identification would be to just start from a leaf node representing an applied identifier, and then search for the subtree that represents the declaration of that identifier. This is however a very slow method, and even though speed is not a priority for our compiler, another method would be preferable. Instead, we can make an identification table, in which we associate identifiers with their attributes. The attribute of an identifier consists of the information that is given at the identifiers declaration.

Scope Levels

The part of the code in which a declaration has effect is called its *scope*, and a *block* is a part of a program that delimits the scope of the declaration within it.

Listing 3.10: Scopes in Imp

```
//scope level 1
global float globalFloat = 3.14;

void Main () { //scope level 2
    { //scope level 3
        { //scope level 4
            int localInt = 2;
            globalFloat = globalFloat + localInt;
        }
    }
}
```

Different languages have different block structures: Monolithic block structure, flat block structure or nested block structure. However, since we want our language to have C-like language (see section 3.1.1), we will be using nested block structure.

A programming language that uses nested block structure allows blocks nested within other blocks, making multiple scope levels possible. See listing 3.10. Declarations in the outermost scope are global declarations. This can be considered as scope level 1. Declarations inside a block are local to that block. Languages that have nested block structures, have the following scope rules [24]:

1. No identifier may be declared more than once in the same block. Nevertheless, the same identifier may be declared in different blocks, even if they are nested.

2. For every applied occurrence of an identifier `I` in a block `B`, there must be a corresponding declaration of `I`. This declaration must be in `B` itself, or (failing that) in the block `B'` that immediately encloses `B`, or (failing that) in the block `B''` that immediately encloses `B'`. In other words, the corresponding declaration is in the smallest enclosing block that contains any declaration of `I`.

When using a nested block structure, the block levels of identifiers are also noted in the identification table.

3.6.2 Type Checking

The contextual analyzer also verifies that the source program does not have type errors. One of the most important features of a statically typed language, is that type errors can be detected at compile-time. This is particularly well-suited for expressions.

An expression can be decorated with a type, so that the type of the expression can be determined, which is useful for checking whether the expression can be used in certain conditions, without actually calculating the result of the expression. For instance, an expression that contains an integer and a floating point value, can be decorated as being a floating point value, since it is a more precise outcome of the expression.

3.7 Error Handling

When compiling a program, the compiler may encounter various errors in the syntax, the phrase structure or the use of types. The developer might for example have misspelled a keyword, forgotten to put a semicolon at the end of a command or assigned a string to an integer variable. When such errors occur, the compiler needs to take steps to either correct the errors, if possible, or provide the developer with a meaningful error report and terminate the compilation. There are different strategies to handle errors and in the following we will describe how we intend to do it in the Imp translator.

3.7.1 Error Correction

Syntactical errors in a program are in most cases typological which result in the program not expressing the wishes of the developer. This could be misspelled function calls, variable names or keywords. Such errors are hard to correct. For example, if a function call is misspelled it can be difficult to analyze if it actually is the function call that is incorrect or if it is the function declaration that is misspelled. Therefore it can almost be impossible to determine if the error is caused by a typographical error (syntactical error) or if the error is a result of the developer not understanding the phrase structure (semantical error). Some semantical errors can be automatically corrected by casting the variables. However, this is not as simple as it seems because it rises the question if the developer actually wants this. Take for instance a computation through recursive method calls where, if a float is cast to an int, the result will accumulate differently than intended.

The Imp translator will not contain any error correction because of the complexity of the problem and because the concern of this project lies elsewhere.

Error Reports

Error reports in Imp should provide the developer with sufficient information to quickly locate and correct the error. For instance, if it is a type error, then the actual and expected type should be provided to enable faster correction by the developer. Meaningful information to include in an error report are listed below.

- The line number on which the error occurred.
- What token was provided as opposed to what was expected for a correct phrase structure.

3.8 Loop Dependence Analysis

As written in the problem statement, section 2.11, we want to achieve implicit parallelism by making loops parallelized automatically. In the Imp programming language these potentially parallel loops are `foreach` loops. Each iteration of a `foreach` in Imp will be executed in parallel threads, if the loop meets certain requirements. These requirement are based on Bernstein's conditions, which are described in section 2.6.

When we use a normal `foreach` loop, we know the order of which the iterations finish in, but in a parallel loop, we do not. This is why it is important that we only allow the `foreach` loops to be parallelized, that give the same result when they are executed in parallel, as when they are sequentially executed. Therefore, we cannot allow a `foreach` loop in which variables with a lower level scope are written to, because we cannot always predict what result those variables will have, when the `foreach` loop is exited. This is in agreement with Bernstein's data dependencies, where one iteration can be viewed as S_1 and another can be viewed as S_2. Whenever either S_1 or S_2 is written to, one of the conditions are applicable, and the loop cannot be parallelized. However, if the variable in question is either declared within the loop, or if it is the `foreach` loop iterator variable, Bernstein's condition's are not applicable, because each iteration will have its own distinct copy of the variable.

Listing 3.11: `foreach` loops in Imp

```
global int globalInt = 0;
void Main () {
    array int [3]intArray;

    // Parallelizable foreach loop
    foreach (int i in intArray) {
        int j = 2;
        i = 40 + j;
    }

    [0]intArray = 0;
    [1]intArray = 2;
    [2]intArray = 5;

    // Unparallelizable foreach loop
    foreach (int i in intArray) {
        globalInt = i;
    }
}
```

As seen in listing 3.11 , we can guarantee that after the first `foreach` loop, all the elements of `intArray` will have the value 42, no matter the order of the iterations, thus it is parallelizable. However, we do not know whether `globalInt` is 0, 2, or 5 when we exit the second `foreach` loop, and thus it is not parallelizable. The same principle goes for function calls within `foreach` loops. A `foreach` loop may only be parallelized, if it does not contain any function calls that assign variables of a lower scope within their function declaration.

This method has a catch nonetheless. It is possible to make some `foreach` loops, in which variables of a lower scope are written to, parallelizable - for instance the `foreach` loop in listing 3.12.

Listing 3.12: Theoretically parallelizable loop

```
global int globalInt = 0;
void Main () {
    array int [3]intArray;

    [0]intArray = 42;
    [1]intArray = 42;
    [2]intArray = 42;

    // this loop is parallelizable in theory
    foreach (int i in intArray) {
        globalInt = i;
    }
}
```

When this loop is exited, we can be certain that `globalInt` will have the value 42, however our method will not allow this `foreach` loop to be parallelized, since a variable of a lower scope is written to within it.

When a situation occurs where a `foreach` loop is nested within another `foreach` loop, then if the outermost loop meets the requirements, described earlier, and the innermost loop does not, then none of the loops may be parallelized. This is due to that, the outermost loop inherits the dependence conditions from the nested loop. So `foreach` loops that contain unparallelizable `foreach` loops, may never be parallelized.

3.9 Code Generation

The syntactical and contextual phases mentioned in sections 3.1.1 and 3.6 only depend on the source program, as these phases analyse the source code for errors. Code generation, on the other hand, both has to do with the source code and its translation to the target language. This section describes how the code generation can be done when developing a new programming language and is based on [24].

As written above, code generation has an extra dependency compared to the syntactical and contextual phase, which makes it difficult to have fixed set of rules when generating code. The reason is that target languages may vary, and the code generator is deeply influenced by its target languages, so you cannot make a universal rule for code generation.

The basic idea of code generation is to convert the AST into a sequence of instructions, which in our case is a sequence of instructions in Java code. Many compilers output assembler code which then is used to generate machine code and finally an executable program. However, since our output language will be a high level programming language and not assembler code we do not have to worry about a certain instruction set for specific hardware or computer architecture. We will simply output a text file containing Java code.

Code generation is able to optimize the sequence of the instructions when generating the code, but since we let the Java compiler handle this, we do not have to worry about in which order the instructions are executed.

The design of the code generator will include the visitor pattern to traverse the AST and output Java code corresponding to the production rule visited. The needed information is taken from the AST objects, e.g. variable names and types.

The code generation should also take care of splitting Imp code into threads. How this task is done, is considered in the following.

3.9.1 Threaded Programming in Java

A thread in a Java program is a path of independent code execution. The most important feature of threads is the possibility to run them concurrently. A Java program consisting of multiple threads is very useful when multiple events or actions needs to occur at the same time. An example could be a computer game that calculates pathfinding while listening for the users input. This could be achieved with multi-threading. The source for this section is [20].

One procedure to create threads in Java is by declaring a class as a subclass of the `java.lang.Thread class`. This is a simple way to achieve concurrency. The run() method of the Thread can be compared with the main() method of a program and is invoked when the thread starts.

Listing 3.13: Class that can be run as a thread

```
class JavaThread extends Thread {
        // Overrides the empty run() method of the java.lang.
           Thread class.
        public void run () {
            // code...
    }
}
```

Listing 3.14: Example of using the class above

```java
        JavaThread t = new JavaThread();
        t.start();
```

Multi-threading can also be achieved using the Runnable interface. This procedure is less simple than the Subclass procedure because it takes a detour around the Runnable interface.

Listing 3.15: Class that implements the Runnable interface

```java
 1  class RunnableDemo implements Runnable {
 2          public void start () {
 3                  // A Thread is created with reference to this
                        class
 4                  Thread thread = new Thread (this);
 5
 6                  // The Thread is started
 7                  thread.start ();
 8          }
 9
10          // Overrides the Runnable run() method
11          public void run() {
12                  // code...
13          }
14  }
```

Listing 3.15 above shows a simple version of this procedure. When the class creates a thread it passes a reference to itself in the constructor. When the thread is started in line 7 it will begin to execute the run() method of the RunnableDemo class starting in line 11.

Which of these two procedures should we use in our project? To decide, we take a look on the main differences between the two. The procedure using the Runnable interface overrides the Runnable run() method and this run() method will have access to all the variables in the Runnable object. It is convenient if the thread takes care of one part of a program, e.g. drawing a graphical output, while still working with the variables of the Runnable object.[14]

In the Subclass procedure however, each thread object will have its own set of variables and not share them with the Runnable object. This is convenient when multiple threads are needed because it will make it easier to distinguish between the different threads. Establishing an idea of what each thread does as an independent object makes this method suitable for multi-threaded programs.

In our project, we plan to run `foreach` loops using multiple threads. This means that we want to have a class for every `foreach` loop. If a program has three different `foreach` loops, the translator will create three thread classes. This means that using the subclass method for creating these class threads will be most advantageous. This is because the thread classes will then be able to contain their own set of variables and not overwrite each others `run` methods.

3.10 Visitor Design Pattern

After an AST has been built for the given source program, we need to specify what tasks to perform when traversing the tree during the various stages of the compilation process. These stages include pretty printing, AST printing, contextual analysis, dependence analysis and code generation. Because these stages all traverse the AST in the same manner, it would be advantageous to devise a way to reuse the traversal mechanism. A possible way to do this is to implement the visitor design pattern. This section describes how the design pattern works and how we would benefit from using it.

The visitor design pattern uses depth-first search to traverse a tree. The visitor design pattern is categorized under a type of design patterns called extension patterns. Extension patterns all focus on simplifying and facilitating extension of code. When extending code in an object-oriented language, like Java, the typical technique is to add methods and/or classes to the object model. Using this technique can sometimes be difficult because the object model can be very complex or the code might not be accessible. However, using the visitor pattern makes it possible for the developer to add support for other developers to extend the behavior of the existing object model without changing it. A possible downside of using the visitor pattern could be that it makes the application more complex than if we just added the methods directly to the AST classes.

The visitor design pattern implements a mechanism called ”double dispatch”. This means that when an object is visited, the selection of which method to execute is based on the type of two objects at the same time. In our AST scenario this means that both the type of the visitor class and the type of the specific AST class determines the proper method to execute.

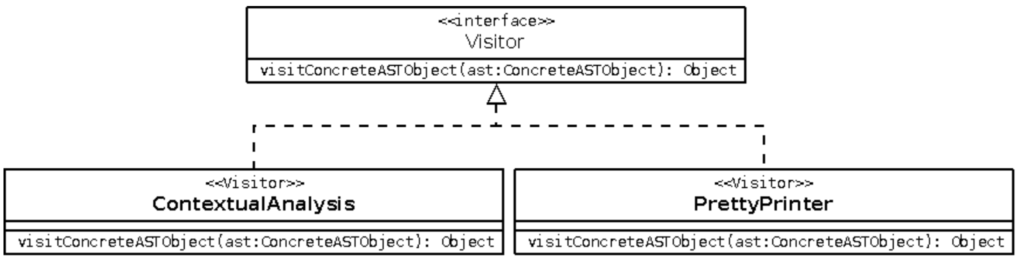
Pros and cons of using the visitor pattern in the Imp compiler.

- Pros
  - All functionality for traversing the AST for a given purpose is gathered in one class.
  - Easy to implement new visitor based classes to traverse the AST.
- Cons
  - More complex than simply just adding methods to AST classes.

3.10.1 Implementation of the Visitor Pattern

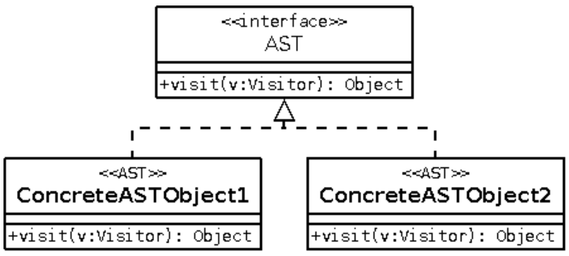
The visitor pattern is implemented by creating an interface class containing all the methods the concrete visitor should implement. For each different purpose of traversing the AST, a new implementation of the visitor interface can be made. See figure 3.9 as an illustration of this concept.

Each concrete AST class implements a visitor method that takes an instance of a visitor implementation as an argument and executes the method in that visitor corresponding to the type of the AST. This concept is illustrated in figure 3.10.



<<interface>>
Visitor
visitConcreteASTObject(ast:ConcreteASTObject): Object
<<Visitor>>
ContextualAnalysis
visitConcreteASTObject(ast:ConcreteASTObject): Object
<<Visitor>>
PrettyPrinter
visitConcreteASTObject(ast:ConcreteASTObject): Object

Figure 3.9: Visitor implementation



<<interface>>
AST
+visit(v:Visitor): Object
<<AST>>
ConcreteASTObject1
+visit(v:Visitor): Object
<<AST>>
ConcreteASTObject2
+visit(v:Visitor): Object

Figure 3.10: AST implementation

3.11 Summary

In this chapter we have discussed several design decisions for the programming language, Imp. We have covered important aspects, such as the syntactical as well as the semantical part of the language. Furthermore, we have reflected on how it is possible to implement threading in Java to take advantage of potential concurrency in the Imp source code and thereby making it parallelizable. Additionally, we have discussed how it is possible to implement the visitor pattern in our implementation and how we are able to draw benefit from it.

Based on the discussions and the considerations we have made, regarding what we want in our implementation, we decided that we want to start by implementing a tokenizer that divides Imp source code into tokens. This makes us able to construct an abstract syntax tree by making a parser. The parser will use the production rules, defined in the CFG, that we have made during the design phase. Furthermore, we are going to make our product such that it satisfies the informal semantics and thereby ensuring that it behaves like it is intended to. Lastly, we are going to implement implicit parallelism in order to satisfy our ambitions with the project.

Chapter 4
Implementation

After having discussed the design issues of Imp, it is time to implement the various parts of the language. This chapter will cover what we have implemented and how it has been done. The chapter contains a description of our implementation of the tokenizer (section 4.1) and how it produces tokens. Additionally, it describes how we implemented our parser (section 4.2) that constructs an AST and how this AST is built up of several classes (section 4.3). Furthermore, this chapter covers our contextual analysis (section 4.5) as well as how we generate Java source code from our Imp source code (sections 4.6 and 4.7).

4.1 Tokenizer

As described in sections 2.2.1 and 3.3, it is necessary to scan through the source program to turn it into tokens. The lexical analyzer in this project is called a tokenizer and this section will describe the implementation of it.

First off, the tokenizer includes an abstract `Token` class, which includes a `name` to identify the specific token and a `lineNumber` to make it easier for the user to find mistakes in his code. Seven classes inherit from the `Token` class, all of which are very similar. The classes `CharToken`, `FloatToken`, `StringToken` (exemplified below) and `IntToken` represent the given value of the token. The `GenericToken` class is used to represent symbols (e.g.: `- + = *` and so on). The `NameToken` represents variable names, the `FNameToken` corresponds to function names and the `TypeToken` represents a type.

Listing 4.1: String token class

```
public class StringToken extends Token{
    public String value;
    public StringToken(String value, int line) {
        super("StringToken", line);
        this.value = value;
```

The primary class is the `Tokenizer`, which takes Imp source code as parameter and contains the scanning algorithm. The class consist of three functions:

- `removeWhiteSpaces`
- `Tokenize`
- `printInput`

The `Tokenize` function starts by creating a double-ended queue for the list of tokens. This is done to ensure that tokens can be obtained from the start of the queue and not just the end. Next, it uses the `removeWhiteSpaces` function to remove any white spaces in the beginning of the source code.

To transform the input into tokens, we made use of the built-in Java packages `Matcher` and `Pattern`. The `Pattern` class is able to represent a regular expression specified in a string, and the `Matcher` class is able to match characters in a given pattern, such as the `Pattern` used in our case. Listing 4.2 illustrates how the packages are used on the `String` token.

Listing 4.2: String regular expression pattern and string matcher

```
...
Matcher stringMatch = stringRegExp.matcher(input);
...
private Pattern stringRegExp = Pattern.compile("\\A\"\\w*\"");
...
```

For each token, a `Matcher` object is used with a `Pattern` object, which by means of regular expressions finds the given token in the input. The regular expressions in `Pattern` were at first difficult to construct, but by looking at the Java documentation we found out how the `Pattern` and `Matcher` syntax were defined. The `Pattern`s created in the tokenizer all make use of the `\\A` -parameter, as seen in listing 4.2. This parameter makes sure that the match is at the beginning of the input, as input are read from the beginning.

If an input matches a token, a new `Token` object is created containing its type and line number and then added to the list of tokens, see listing 4.3 below.

Listing 4.3: Adding String token to tokenList

```
if (typeMatch.find()) {
    String type = typeMatch.group(0).trim();
    Token token = new TypeToken(type, lineNumber);
    tokenList.add(token);
```

The `typeMatch.group()` gives the possibility to divide regular expressions into groups. The `group(0)` is the entire regular expression. The `trim()` command removes whitespaces. The last function, `printInput`, in the `Tokenizer` class allows the output of the tokenizer to be printed to the console for debugging purposes.

4.2 Parser

The parser has been implemented by adding functions for all the production rules in the language grammar, that is, the CFG (see appendix A) has been translated into actual code. As explained in section 2.2.1, the parser takes the list of tokens, generated by the tokenizer, and uses this list to check if the source program respects the language grammar defined in the CFG.

Since the parser for the Imp language is a recursive descent parser, described in section 3.5, it constructs an AST by parsing from the top. The parser starts with `parseProgram`, then proceeds to `parseDeclaration` or `parseFunction`, and so forth, depending on the source program. The `parseCommand` function is taken as an example and will be described briefly. Besides the production rule functions, there are two important functions in the parser, namely:

- `accept`
- `remove`

To check for the correct and expected tokens in the source program, the `accept` function takes in the expected token as a string and compares it with the first token in the `tokenList`. The function returns `true` in case of equality with the expected token and `false` if no such element exists. The `remove` function removes the first token in the `tokenList`, because it is always the first token in the list which is checked.

Listing 4.4: `parseCommand()` function

```
1    Command parseCommand() {
2        Command commandAST = null;
3        SingleCommand singleCommandAST = null;
4        debugMessage("Trying to parse command");
5        singleCommandAST = parseSingleCommand();
6        commandAST = new NonSequentialCommand(singleCommandAST);
7        if (accept("NameToken", "if", "while", "{", "TypeToken",
             "return", "FNameToken")) {
8            commandAST = new SequentialCommand(singleCommandAST,
                 parseCommand());
9        }
10       return commandAST;
11   }
```

The `parseCommand` function in listing 4.4 creates a `SingleCommand` object by visiting the `parseSingleCommand()` function. Before doing this, a `debugMessage` prints the current job of the parser. This message makes it easier to find errors in the parser. The CFG for `Command` states that it is possible to have more than one `SingleCommand`, see listing 4.5 below. So, the parser needs to check if tokens from another `SingleCommand` are present. The `accept` function described above takes care of this on line 7 in listing 4.4, and if the right token is next in the list, a new `Command` object will be created by calling `parseCommand()` recursively. This is basically how all the parse functions are built up, with several if-statements however. Using a switch statement would have been more readable, unfortunately the Java language did not allow us to compare strings (the name of the token) in switch statements, hence the use of nested if-statements.

Listing 4.5: Command production rule from the CFG

```
Command ::= Single-command
       |    Single-command Command
```

4.3 Abstract Syntax Tree

The following elaborates on how we implemented the AST, exemplified by using the declaration of a string variable. As described in section 3.4, we need to store our source program in an AST to complete the rest of the compilation phases. The AST is built when the parser goes through the source program which at this phase in the compilation process is a list of tokens supplied by the tokenizer. The parser uses the production rules from the CFG to generate the objects that make up the AST for the entire source program. For most production rules there are corresponding AST objects, where some have several constructors. The following figures show this relation for the `Program` (figure 4.1) and `Declaration` (figure 4.2) production rules.

- Program ::= ((global Declaration))* Function

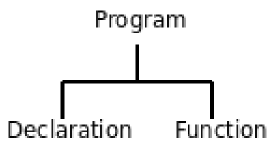


Figure 4.1:

- Declaration ::= Type-denoter Name = ((String-literal | Expression)) ;
  | const Type-denoter Name = Literal ;

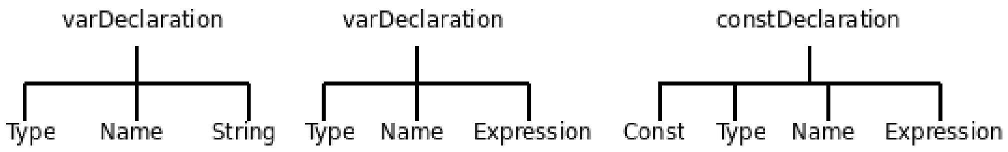


Figure 4.2:

As illustrated in figure 4.2, there are two different AST classes for a declaration depending on whether it is a constant or a variable declaration assigning a string or an expression. The first class is the `constDeclarationAST` which corresponds to a constant declaration, the second, `varDeclarationAST`, contains two constructors; one for a string and one for an expression.

4.3.1 Class Representation

All AST classes inherit from the `AST` class. This is done to get an object-oriented structure and to ensure that all classes include a `visitor` method. Each AST class has properties that suit the elements of a specific production rule. For example, the `Program` class has properties for declarations and functions. The `Declaration` and `Function` then have properties for other AST classes, and this is how the tree is structured. Figure 4.3 is an excerpt of the inheritance amongst AST classes.

The AST is built by the parser by going through the list of tokens and creating AST objects corresponding to the source code, giving an error message if it is ill-formed.

4.4 AST Printer

To get an idea of how our parser created the AST, and if it was correct, we needed some form of representation of it. A good solution seemed to be a simple visual representation of the AST. This meant we had to have some way of drawing the tree, and to save time we used a graph visualization tool called Graphviz. Graphviz is an open source package that can draw graphs and diagrams in many formats by using simple text files describing the graphs[1]. The way we used this, was to have a class designed to write such a text file and then use a command-line tool called `dot` from the Graphviz package to draw a graph, based on the text file. To explain how this works, we will start with a ”Hello World” program written in Imp and end with a graphical version of the programs AST.

Listing 4.6: Imp Hello World program

```
void Main() {
    Print("Hello World");
}
```

The AST printer uses the following two classes:

- The `GraphRepresentation` class handles the text file, which `dot` uses to draw a graph. It has a string buffer that contains the content to be written to the text file.

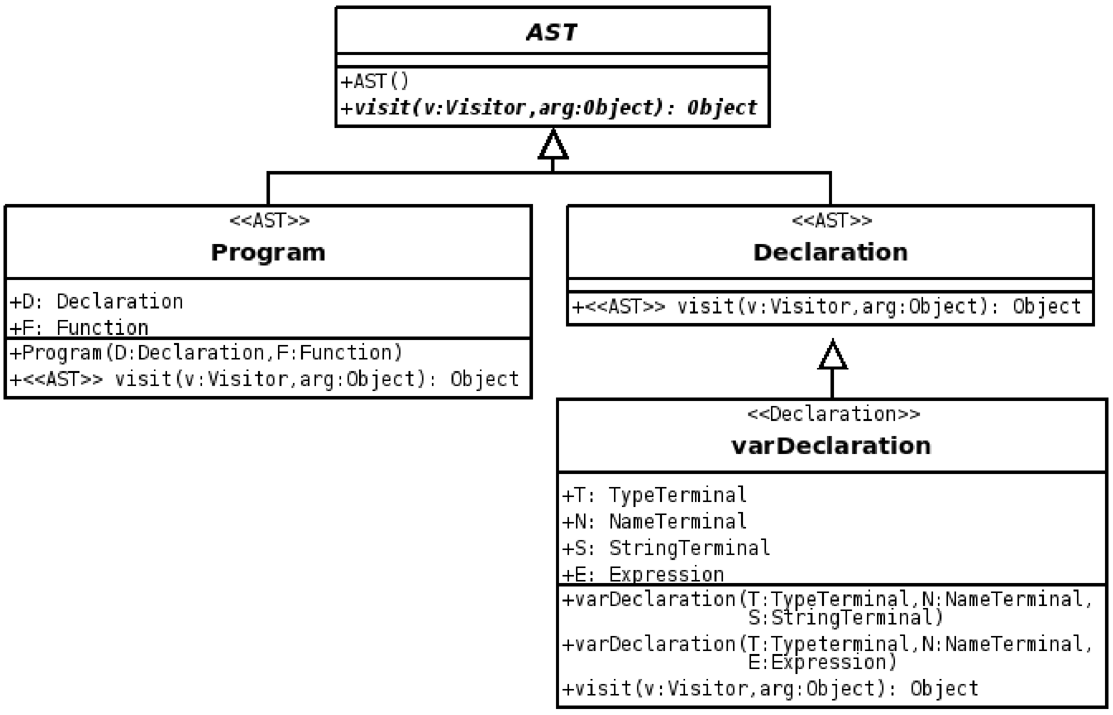

AST
+AST()
+visit(v:Visitor,arg:Object): Object
<<AST>>
Program
+D: Declaration
+F: Function
+Program(D:Declaration,F:Function)
+<<AST>> visit(v:Visitor,arg:Object): Object
<<AST>>
Declaration
+<<AST>> visit(v:Visitor,arg:Object): Object
<<Declaration>>
varDeclaration
+T: TypeTerminal
+N: NameTerminal
+S: StringTerminal
+E: Expression
+varDeclaration(T:TypeTerminal,N:NameTerminal, S:StringTerminal)
+varDeclaration(T:Typeterminal,N:NameTerminal, E:Expression)
+visit(v:Visitor,arg:Object): Object

Figure 4.3: Excerpt from the AST class diagram.

The class has functions for adding new nodes, leaves and relations between these, by inserting strings to the string buffer. Finally, a function will take the string buffer and write it to a text file with the .gv extension, execute the `dot` program and give it the text file as input. The `dot` program will then draw the AST as a graph and output it to a new file.

- The `Printer` class uses the `GraphRepresentation.java` to output a graph. It uses the visitor pattern to traverse the AST, and while doing so it calls the functions of the `GraphRepresentation.java` class to add new nodes, leaves and relations. Everytime an object is visited, new nodes are created to represent its children, and relations from the current node to the children notes are added. The children of the object is then visited. Listing 4.7 below gives an example of how a node, its child and a relation between them is added.

Listing 4.7: part of the `Printer.java` AST printer

```
1 GR.addChild(ast.getClass().getSimpleName(), ast.toString());
2 GR.addChild(ast.F.getClass().getSimpleName(), ast.F.toString());
3 GR.addRelation(ast.toString(), ast.F.toString());
4 ast.F.visit(this, null);
```

The preceding code snippet is taken from `visitProgram()`. Since this is always the first object visited, this will add a node representing itself in line 1 and then add the function AST object in line 2. Line 3 is where the relation from the program object to the function object is created. Finally, line 4 will visit the function object and continue adding to the string buffers of the `GraphRepresentation.java` class. However, if the visited object is a terminal a leaf will be added. This is drawn in another way than other nodes and therefore requires a different output to the .gv file. Listing 4.8 shows how a leaf with its relation to a given parent is added.

Listing 4.8: Part of the `Printer.java` AST printer

```
GR.addLeaf(ast.name, ”unique”+unique);
GR.addRelation(ast.toString(), ”unique”+unique);
unique++;
```

When all objects have been visited, the function that writes the file and executes the `dot` program will be invoked. This is done in the `visitProgram()` function of the `Printer.java` class. A snippet of the output text file looks like this:

Listing 4.9: Part of the .gv file

```
digraph G {
node [shape = box, fontname=”Times”];
node [label=”Program” ] ”AST.Program@1a679b7”;
node [label=”Function” ] ”AST.Function@80f4cb”;
...
”AST.Program@1a679b7” -> ”AST.Function@80f4cb”
...
}
```

All other nodes except `Program` and `Function` have been omitted but it is still clear that two nodes are created and a relation between them is defined using the ”–>” in line 6.

When the .gv file has been given to the `dot` program as input file, the resulting graph will be the one seen in figure 4.4. This visual representation has proven to be very useful in both getting an idea of how the parser builds the AST for the program and in debugging different parts of the translator.

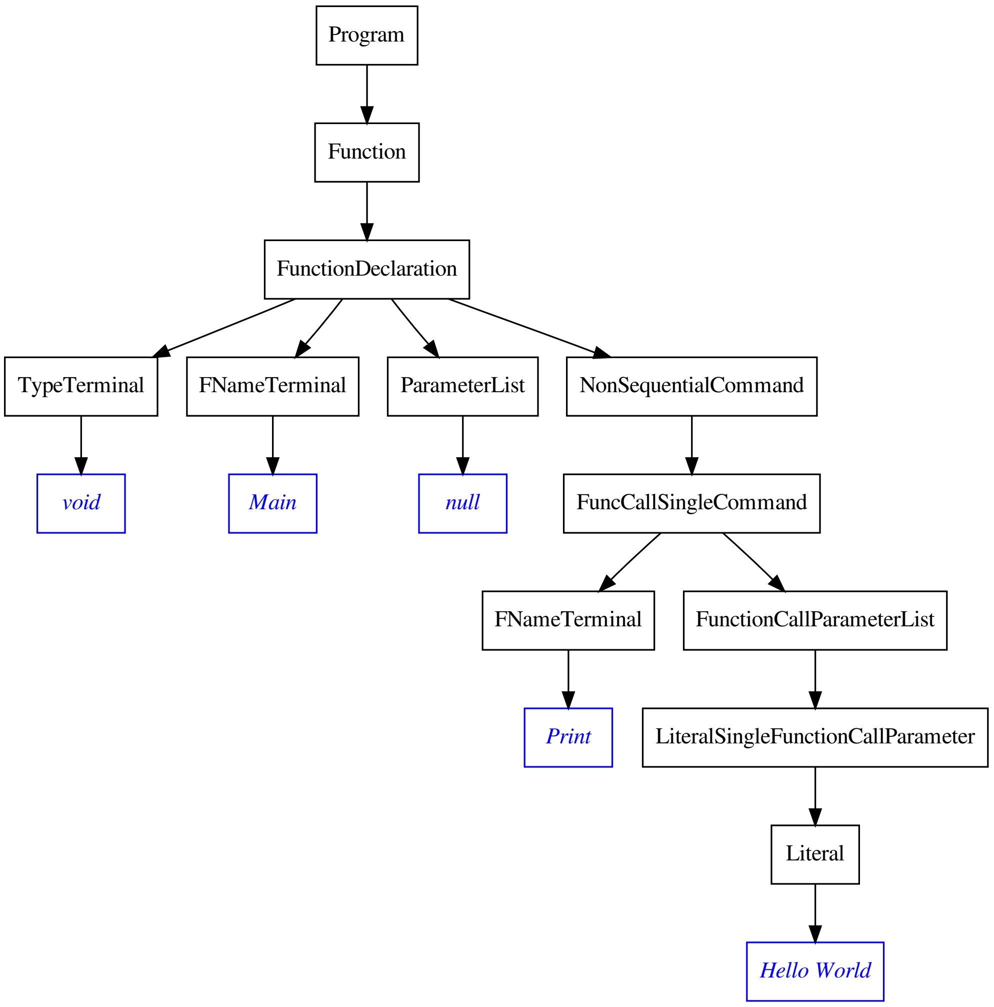


Figure 4.4: Hello World Abstract Syntax Tree.

4.5 Contextual Analysis

In this section, we describe how we implemented the contextual analysis of our language, which consists of type checking and scope rules as explained in section 2.2.2.

4.5.1 Type Checking

Type checking is done by making use of an identification table, as written in the design of the contextual analysis, sections 3.6.1 and 3.6.2.

To discover the types in Imp, the AST needs to be traversed. Each visit-method in the visitor pattern is able to return an object, which we use to return the type. If the visited AST object has a type, we can check for type consistency. This is fairly easy to do with literals, but with variables we have to fetch the declaration that defines the variable name and type, and check for type errors. However, not every AST object returns a type, e.g. the object that represents a while construct does not have a type.

Listing 4.10: Example of type checking in Imp

```
 1  public Object visitSequentialTerm(SequentialTerm ast, Object o)
       {
 2          ast.T1.visit(this, null);
 3          ast.T2.visit(this, null);
 4
 5          Type newType = ast.T1.type.checkTypes(ast.T2.type);
 6
 7          if (newType == null) {
 8              errorMessage(ast.O.line, ”Type mismatch”);
 9          }
10          ast.type = newType;
11
12          return newType;
13  }
```

Listing 4.10 above shows how the checker can both check the types of the child nodes and return the nodes type. Lines 2 and 3 visit the two child nodes. These nodes decorate themselves with their types and return them. Afterwards, we check if they match each other according to our type rules (line 5). If they match, the checker returns the type. If they do not match, an error message is printed to the user and the translation aborts (line 8).

4.5.2 Scope Rules

As described in section 3.6, we have decided to use nested scope rules, the rules of which can be found in the same section. To implement this, we use a list that keeps track of all the declarations and their scope levels.

In the tree-traversal process, we insert a declaration in the identification table every time a declaration is encountered. If there is no other declaration with the given name in the scope level, the insertion is done. Else, the user is notified of the error in the source code and the translation is aborted.

4.6 Loop Dependence Analysis

In order to analyze the input program for loops and their content, the AST is traversed, using the visitor design pattern. At this point in the compilation process, the AST is decorated with useful information that can used in the dependence analysis.

Before code for parallel `foreach` loops can be generated, we need to determine if it is possible. As explained in section 3.8, a `foreach` loop can be parallelized, if it does not write to variables of a lower level scope. This is handled with a special parameter, that is sent with every node in the subtree under the `foreach` loop called `ParaArg`. The variables that the `ParaArg` class contain can be seen in listing 4.11, which is elaborated in the following.

Listing 4.11: ParaArg variables

```
...
public boolean canBeParallelized = true;
private ArrayList<NameTerminal> allowedNames = new ArrayList<
    NameTerminal>();
private ArrayList<NameTerminal> usedArrayNames = new ArrayList<
    NameTerminal>();
public NameTerminal arrayName;
...
```

`Nameterminals` of variables, that are allowed to be writable, within a `foreach` loop are added to the `allowedNames` arraylist. This includes the `foreach` loop iterator variable, and all variables declared within the loop. Whenever a variable is written to within the loop, its `NameTerminal` is compared to the `allowedNames` list, and if the name is not on the list, the `canBeParallelized` boolean is changed to false. That is, the given foreach loop is not parallelizable.

The same analysis is performed on function declarations, so it can be determined if they can be used in parallel `foreach` loops. All the function declarations are decorated accordingly.

Listing 4.12: The visitForeachSingleCommand method

```
...
public Object visitForeachSingleCommand(ForeachSingleCommand ast
    , Object o) {
     ParaArg arg = new ParaArg(ast.N1, ast.N2);
     ast.C.visit(this, arg);

     if (arg.matchesInUsedArrayNames(ast.N2)) {
          arg.canBeParallelized = false;
     }
     if (o instanceof ParaArg) {
          ParaArg parentArg = ((ParaArg)o);
          parentArg.addUsedArrayName(ast.N2);
12          parentArg.getUsedArrayNames().addAll(arg.getUsedArrayNames());
13      }
14
15      if(o instanceof ParaArg && !arg.canBeParallelized)
16          ((ParaArg)o).canBeParallelized = false;
17      ast.isParallel = arg.canBeParallelized;
18      if (ast.isParallel) {
19          System.out.println("foreach loop at line "+ast.N1.line+" is parallelizable");
20      }
21          else
22              System.out.println("foreach loop at line "+ast.N1.line+" is not parallelizable");
23      return null;
24  }
25  ...
```

`Foreach` loops can only be parallelized if their nested `foreach` loops also can. This is handled in lines 15 and 16.

The `usedArrayNames` list and `arrayName` variable, are used in special situations, where a `foreach` loop is nested within another `foreach` loop, and they both iterate over the same array. In such a scenario, only the innermost `foreach` loop may be parallelized. This is because we cannot allow there to exist concurrent copies of the same array. We cannot determine if the elements of the array will have the same values, as if they were sequentially iterated.

4.7 Code Generation

The following section will describe how we have implemented code generation in the translator. It will contain the discussions and thoughts we have made throughout the development process of our code generation part of the product.

Again, we have been able to draw benefit from the visitor pattern to traverse through the decorated AST. With Java being our target language, the main challenge is to convert the information from our AST into working Java code. Several aspects must be taken into consideration in order to perform this transformation. We need to make sure that the contextual elements of Imp are compatible with the contextual elements of Java.

One of the first questions that arose in relation to code generation was how we were going to do the actual code generation based on our AST. Since we were all familiar with the Java language, this conversion-phase was not as problematic as we had expected. Below is a piece of code that converts while-statements from the AST (from Imp) to Java:

Listing 4.13: `visitWhileSingleCommand` function

```
public Object visitWhileSingleCommand(WhileSingleCommand ast, Object o) {
    String tmp = "while (" + ast.B.visit(this, null) + "){\n";
    tmp += ast.C.visit(this, null) + "}\n";

    return tmp;
}
```

The code simply gathers the required information from the AST and returns a string containing Java source code with the relevant information.

Another interesting example is how we convert the declaration of variables:

Listing 4.14: `visitVarDeclaration` function

```
public Object visitVarDeclaration(VarDeclaration ast, Object o)
    {
    String tmp = "";

    if (o != null)
        tmp += "public static ";

    tmp += (String)(ast.T.visit(this, null)) + ast.N.visit(this,
        null);

    if (ast.E != null)
        tmp += "= " + ast.E.visit(this, null) + ";\n";
    else if (ast.S != null)
        tmp += "= " + ast.S.visit(this, null) + ";\n";
    return tmp;
}
```

In code example 4.14, it is shown how the visitor pattern is used to traverse the subelements of the AST. The `visitVarDeclaration` is a good example of how we obtain information from the subelements by using the visitor pattern. We invoke a visit function and get a string which, combined with additional information, ends up being an actual line of Java code.

4.7.1 Generating the Code for Implicit Parallelism

This section will be based on the following Imp code:

Listing 4.15: An Imp foreach loop

```
void Main() {
    array int [3]myarray;
    [0]myarray = 0;
    [1]myarray = 1;
    [2]myarray = 2;
    foreach (int x in myarray) {
        x = x + 1;
        Printi(x);
    }
    Printi([2]myarray);
}
```

This program in listing 4.15 defines an array in line 2 and enters data into the cells on lines 3 to 5. The `foreach` loop will then go through the array and add 1 to all the cells and finally print it. To check if the array has been changed, the last cell will be printed outside the `foreach` loop.

When the CodeGen reaches the `foreach` loop, the output will depend on whether the `foreach` loop is parallelizable or not. If it is not, the generated code will be a Java `for` loop where the loop counter will increment until reaching the end of the array. This will allow the loop and functions to read from and write to local variables. The `foreach` loop in line 6 only writes to the current cell in the array, and not to any other variables, and can therefore be parallelized.

When a `foreach` loop is parallelizable, the code generation is a bit more extensive. The idea is that each iteration of the `foreach` loop will be executed in its own thread. This means that a `foreach` loop running on an array with length 10 will create 10 independent threads. This is done by first creating a new class file that will act as the code within the `foreach` loop would do. The thread class will have a constructor where variables are passed to the class. The translator will take all variables and arrays available in the current scope level and pass it to the class. However comprehensive this might seem, it allows variables defined outside the `foreach` loop to be read from inside the `foreach` loop. The thread class will be created using the method described in 3.9.1 meaning that the class will have its own instances of the variables passed to it. The `run()` method of the thread class will contain the code that originally was contained in the `foreach` loop.

Listing 4.16: The `ForeachThread#.java` thread class

```
class ForeachThread1 extends Thread {
    private int _i1;
    private int[] myarray;
    ForeachThread1(int _i1, int[] _myarray){
        this._i1 = _i1;
        myarray = _myarray;
    }
    public void run() {
        myarray[_i1] = myarray[_i1] + 1 ;
        ImpProg.Printi (myarray[_i1] );
    }
}
```

Code example 4.16 shows the class created for the `foreach` loop. All the variables are declared and then initialized in the constructor. The `run()` method contains the actual code from the `foreach` loop. It is important to notice that the `"x"` from the source code 4.15 will be renamed in the thread class. This is done because the whole array is actually passed into the constructor. The `"x"` is always renamed to `"myarray[_i1]"` where `"_i1"` is the iterator in the `foreach` loop, so to speak. This is easier to understand when looking on lines 8 to 11 in listing 4.17 where the thread classes are created. Here "`_i`", the iterator from the `for` loop is used to construct the thread objects. This way the iterator from the `for` loop controls what cell in the array the `foreach` loop can write to.

In the main program class file, the `foreach` loop code consists of two important `for` loops. The code for the `foreach` loop starts in line 6 and ends in line 22.

Listing 4.17: part of the `ImpProg.java` codegen output class

```
 1 public static void Main (){
 2     int [] myarray  = new int [3 ];
 3     myarray [0 ] = 0 ;
 4     myarray [1 ] = 1 ;
 5     myarray [2 ] = 2 ;
 6     // This array contains the threads
 7     java.util.ArrayList<ForeachThread1> looplist1 = new java.
           util.ArrayList<ForeachThread1>();
 8     for (int _i = 0; _i < myarray.length ; _i++) {
 9         looplist1.add(new ForeachThread1(_i , myarray));
10         looplist1.get(looplist1.size()-1).start();
11     }
12     //This loop makes sure threads are finished before
           continuing
13     for (int i = 0; i < looplist1.size(); i++) {
14         if (looplist1.get(i).isAlive())
15         {
16             try {
17                 looplist1.get(i).join();
18             } catch (InterruptedException e1) {
19                 e1.printStackTrace();
20             }
21         }
22     }
23     ImpProg.Printi (myarray [2]);
24 }
```

The first `for` loop, starting in line 8, uses an `arraylist` to contain the thread class objects. The threads are first constructed with the needed variables and added to the array, then started with the `start()` method in line 10.

The second loop makes sure that all threads are finished before the main program continues. If this was not implemented, programs could not be guaranteed to have the intended operation. By waiting for all threads to finish, the program will have the same outcome no matter if had been parallelized or not.

The output for the program in listing 4.15 is expected to be the following numbers: 1,2,3,3. The order of the first three varied when running the program, for instance we encountered the following output: 1,3,2,3. This is an indication that the numbers are printed from different threads, and that these threads do not necessarily finish in the same order as they are started.

4.8 Summary

This chapter accounted for how the Imp language has been developed. Section 4.1 and 4.2 covered how the source code is recognized as tokens by means of regular expressions. The tokens are then used to create an AST. How the AST and its print class was implemented is explained in sections 4.3 and 4.4. Having the source code represented as an AST, allowed us to use the visitor design pattern in the preceding steps of the translation. The type checking and scope rules are implemented in 4.5 by using an identification table. The parallelization of the Imp source code is taken care off in 4.6 by analyzing `foreach` loops and deciding if they are parallelizable. Finally, the generation of Java code from Imp code is elaborated in 4.7. To uphold the parallelism in Java, the code generator utilizes multiple threads.

Chapter 5
Recapitulation

This chapter contains our reflections (section 5.1) on our development process, as well as the final outcome of our product. It also contains our conclusion (section 5.3) of our project as well as potential future work (section 5.2)if additional time had been available.

5.1 Reflections

At the beginning of our project, we decided that we wanted to use certain development methods (see the Approach chapter) in order to ensure a steady development process. How did these development methods perform in practice? Daily standing meetings were meant to increase the efficiency of our meetings. We used this method at first, but we did not find it as effective as first assumed, therefore we skipped them midway through our development process. Additionally, we decided that we wanted to try to use a project management support tool, called Trac. Unfortunately this seemed to be a bad solution for a relatively small project as ours, since coordination and task tracking could easily be handled with the use of post-it notes. A post-it with the name of a task, was put on one part of the wall, if it was not completed, and another place if they were done. Each group member would take a post-it note, when a task was given to them.

What went well in our development phase? We put up several major milestones for the development of our product, which turned out to be a good solution, since we had concrete goals to reach within a certain timespan. Another useful tool that we used in this project was pair programming. A lot of errors were caught during development. It gave us the opportunity to discuss solutions to smaller problems before implementation, which seemed to increase the quality.

What did we actually achieve in this project? First of all, we made Imp - an imperative implicit parallel language. The overall outcome of the programming language itself is what we hoped for it to be. It is not the next major programming language, but it is well suited for illustrating concepts of programming language development and implicit parallelism. Furthermore, we have constructed a translator which is able to convert parts of the sequential imp source code into parallel Java source code. The making of this project has provided us with a thorough understanding of how programming languages and their language processors work and how they are developed.

Another question that inevitably rises is what could have been better? Perhaps smaller milestones could have encouraged the group to work even more intensely towards the larger goals if we had divided them into smaller ones. This would have given a better overview of the individual tasks that arise while providing a better overall project structure.

5.2 Future Development

We had a limited timespan and was forced to confine implicit parallelism down to `foreach` loops. If more time had been available, we could have been able to continue the development of Imp and expand its code constructs that could take advantage of implicit parallelism.

Some of the features we would like to have included are:

- Implicit parallelism on independent declarations of variables and applied occurrences of them.
- Improve and correct the grammar to ease the use of the language, e.g. function calls in expression, declarations of negative integers and floats, function calls in variable declarations.
- Type casting.
- Option to include custom libraries.
- Concatenation of characters and strings.
- Option to compile to Java byte code, or possibly assembly.

Additionally, if we had implemented more implicit parallel features in our language, we may potentially have obtained faster execution time as well as moved even more responsibility away from the programmer.

5.3 Conclusion

At the beginning of the project, we wondered if the translator could assist with the development of multi-threaded programs. Furthermore, we wanted to know if we could automate the process of translating sequential source code into concurrent code, and thereby moving the responsibility from the developer to the translator, as stated in the initiating problem.

To answer this question, we needed to analyze the various aspects of language processors (being its compilation phases, section 2.2), analysis of potential dependencies (section 2.6) and various topics related to programming languages. From the experiences gained in the analysis, we started to design the programming language, Imp.

The Imp translator is made up by a tokenizer (section 4.1) which produces the tokens and a parser (section 4.2) which applies the production rules and thereby outputs an abstract syntax tree. Furthermore, our checker (section 4.5) performs a contextual analysis and decorates our abstract syntax tree. Dependence analysis is also performed to determine which `foreach` loops can be parallelized. Then code generation (section 4.7) is performed and Java code is outputted. These steps are designed and implemented by using the theory from our analysis chapter and thereby enabling us to check whether code can be parallelized.

Another aspect of Imp was that we wanted it to be easily readable as a main priority. This also justifies our choice of wanting a language with a C-like syntax. A side-effect of wanting a relatively small and simple language was high orthogonality, mainly because the programmer has limited, yet precise ways of expressing himself. These properties satisfies our language evaluation criteria (section 2.10).

The main question still remains unanswered, therefore we need to look into what the translator produces to be able to conclude if we managed to move the responsibility to the translator while obtaining implicit parallelism. In section 3.9 we have shown how Imp source code is translated to Java source code. This section (3.9.1) also illustrates that the translator is able to translate sequential source code to parallel code by making use of threads and thereby providing implicit parallelism to the programmer in our `foreach`-constructions. Furthermore, this also means that we have shown that it is possible to move the responsibility from the programmer to the translator, and thereby satisfying the initial problems of our analysis.

Bibliography

[1] Graphviz - Graph Visualization Software.
`http://www.graphviz.org`.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change.* 1999.

[3] Elliot Berk. Jlex: A lexical analyzer genarator for Java.
`http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html`, 1997.

[4] Calin Cascaval Christoph von Praun, Luis Ceze. Implicit Parallelism with Ordered Transactions.
`http://www.cs.washington.edu/homes/luisceze/publications/ipot˙ppopp07.pdf`, 2007.

[5] Community. Haskell.org.
`http://www.haskell.org/`.

[6] Devin Cook. Lookahead Left to Right Parsing (LALR).
`http://www.devincook.com/goldparser/doc/about/lalr.htm`, 2008.

[7] Keith D. Cooper and Linda Torczon. *Engineering a compiler.* Morgan Kaufmann, 2008.

[8] Oliver Enseling. Build your own languages with JavaCC.
`http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-cooltools.html?page=1`, 2000.

[9] Lars Marius Garshol. BNF and EBNF: What are they and how do they work?
`http://www.garshol.priv.no/download/text/bnf.html`, 2008.

[10] Prof. Dr. Michael Gerndt. Parallel Programming - Data dependence.
`http://www.lrr.in.tum.de/~gerndt/home/Teaching/SS2007/ParallelProgramming/03DataDependence.pdf`, 2009.

[11] Scott Hudson. LALR Parser Generator for Java.
`http://www.cs.princeton.edu/~appel/modern/java/CUP/`, 1999.

[12] Hans Hüttel. *Pilen ved træets rod.* Aalborg Universitet, 2009.

[13] Yuan Lin. *Concurrency vs Parallelism, Concurrent Programming vs Parallel Programming.* 2006.

[14] Clark S. Lindsey. Introduction to Threads.
`http://www.particle.kth.se/~lindsey/JavaCourse/Book/Part1/Java/Chapter08/threads.html`, 2008.

[15] Sun Microsystems. Project Fortress.
`http://projectfortress.sun.com`.

[16] Computation Structures Group MIT. pH : Parallel Haskell.
`http://csg.csail.mit.edu/projects/languages/ph.shtml`.

[17] Gordon E. Moore. Cramming more components onto integrated circuits.
`ftp://download.intel.com/museum/Moores˙Law/Articles-Press˙Releases/Gordon˙Moore˙1965˙Article.pdf`, 1965.

[18] Dr. Ralf-Peter Mundani. Parallel computing.
`http://www.inf.bv.tum.de/lehre/parallel/`, 2009.

[19] Rishiyur S. Nikhil and Arvind. *Implicit Parallel Programming in pH*. 2001.

[20] Inc O'Reilly & Associates. Exploring java: Threads.
`http://oreilly.com/catalog/expjava/excerpt/index.html`, 2001.

[21] Kurt Nørregård. Functional Programming in Scheme.
`http://www.cs.aau.dk/~normark/prog3-03/html/notes/theme-index.html`, 2008.

[22] Robert W. Sebesta. *Concepts of programming languages*. Pearson Education, 2006.

[23] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 5. edition edition, 2008.

[24] David A. Watt and Deryck Brown. *Programming languages processors in Java*. Prentice Hall, 2000.

[25] Andy Wellings. *Concurrent and Real-Time Programming in Java*. 2004.

Appendices

Appendix A
Context Free Grammar

This chapter contains the CFG, written in EBNF, for the Imp programming language. As written in section 3.1.1, the language is meant to be very similar to C, as we found C's readability very good. This can also be seen by the choice of terminals and non-terminals, which are defined in the next sections.

A.1 Terminals

`; { } = ( ) + - * / if else while const return == > < != <= >= true false space tab end-of-line`

A.2 Non-Terminals

`Program Function Function-declaration Parameter Parameterlist Function-call Function-call-parameter Function-call-parameterlist Parameter Command Single-command Expression Single-expression Declaration Type-denoter Name FName Identifier Integer-literal String-literal Char-literal Literal Comment Letter CapLetter SmallLetter Digit Graphic`

A.3 Production Rules

This section holds the production rules, explained in 2.2.1 and designed in 3.1.1, for the Imp language. It is written in EBNF, which extended uses are briefly described below:

- Double parentheses, (( )), are meant as part of the EBNF, whereas single parentheses are part of the Imp language.
- The $*$ means none or many of a given rule.
- Double square brackets, [[ ]], denote zero or one appearance of a rule, whereas single square brackets are part of the Imp language.
- The $+$ means one or more of a given rule.

Listing A.1: Context Free Grammar in Extended Backus-Naur Form

```
Program ::= (( global Declaration ))* Function

Declaration ::= Type-denoter Name =(( String-literal |
    Expression )) ;
    |   const Type-denoter Name = Literal ;
    |   array Type-denoter Name[ Expression ] ;

Function ::= (( Function-declaration ))+

Function-declaration ::= Type-denoter FName ( Parameterlist ) {
    Command }

Parameter ::= [[array]] Type-denoter Name
    | [[array]] Type-denoter Name , Parameter

Parameterlist ::= [[ Parameter ]]

Function-call-parameter ::= Name , Function-call-parameter
    |   Literal , Function-call-parameter
    |   Name
    |   Literal

Function-call-parameterlist ::= [[ Function-call-parameter ]]

Command ::= Single-command
    |   Single-command Command

Single-command  ::= FName ( Function-call-parameterlist );
    |   Name = (( FName ( Function-call-parameterlist ) |
          Expression ));
    |   [expression]Name = (( FName ( Function-call-
          parameterlist ) | Expression ));
    |   if ( Bool-expression ) { Command } [[else { Command }]]
    |   while ( Bool-expression ) { Command }
    |   foreach (Type-denoter Name in Name) { Command }
    |   { Command }
    |   Declaration
    |   Returner ;

Returner ::= return [[ Name | Literal ]]

Expression  ::= Term
    |   Term (( (( + | - )) Expression

Boolean-expression  ::= Single-boolean
```

```
    | (( Single-boolean  (( (( '||' | && )) Boolean-expression ) ) ))

Single-boolean ::=  Factor (( == | > | < | != | <= | >= )) Factor
    | true
    | false

Term ::= Factor (( (( '*' | / )) Factor ))*

Factor ::= ( Expression )
    |   Name
    |   Float-literal
    |   Int-literal
  |   [Expression]Name

Type-denoter ::=     Identifier

Name ::= SmallLetter Identifier

FName ::=   CapLetter Identifier

Identifier ::=  Letter (( Letter | Digit ))*

Integer-literal ::= Digit
    |   Digit Integer-literal

Float-literal ::= Integer-literal.Integer-literal

String-literal ::=  '' Graphic* ''

Char-literal ::= '((Graphic))'

Literal ::=      Integer-literal
    |   Float-literal
    |   String-literal
    |   Char-literal

Comment ::= // Graphic* eol

Letter ::=  SmallLetter
    |   CapLetter

CapLetter ::= 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

SmallLetter ::= 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

```

```
Digit ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Graphic ::= Letter | Digit | space | tab | . | : | ; | , | ~ |
    ( | ) | [ | ] | { | } | _ | ! | ' | ` | '' | # | $ | %
```

Appendix B
Abstract Syntax Tree

Abstract syntax tree generated from mySmallProgram.imp source code (see D.7).

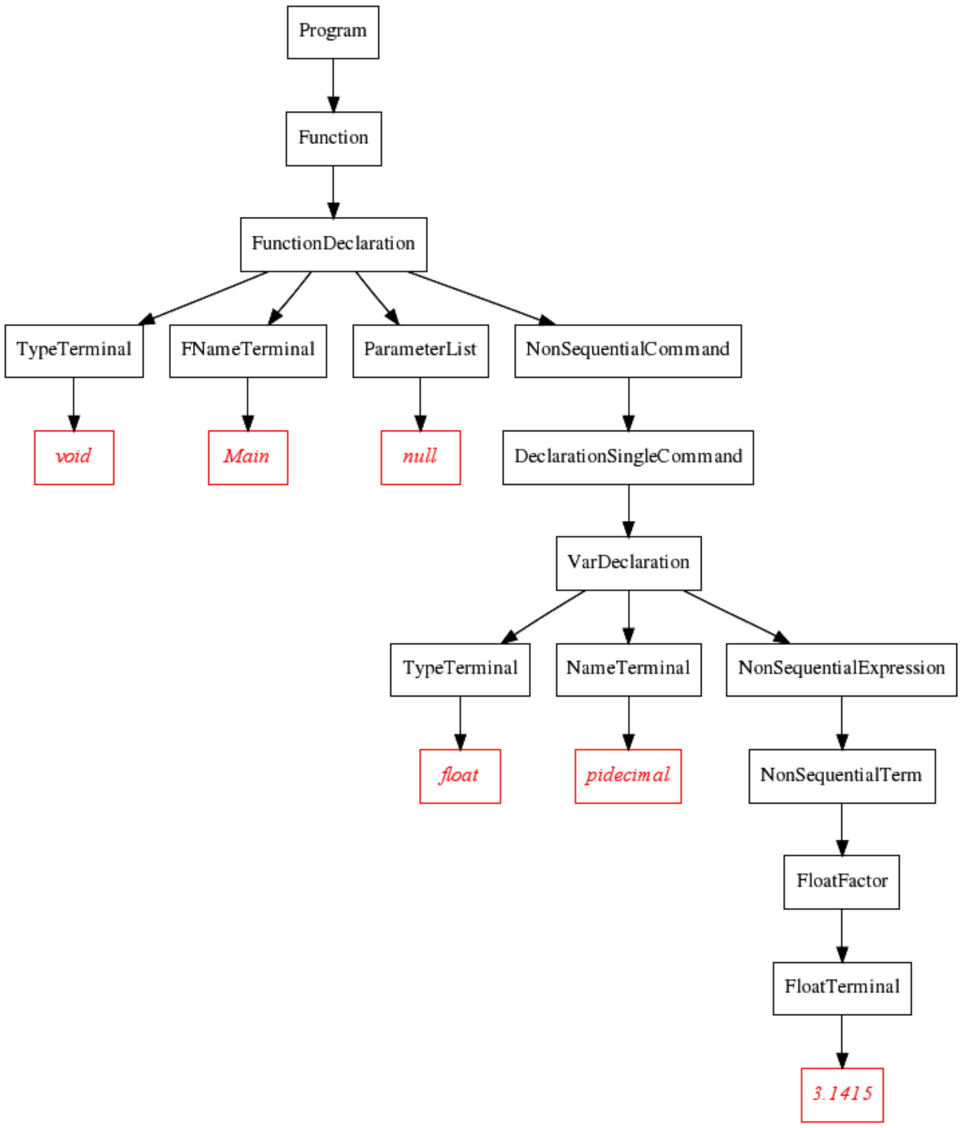


Figure B.1: A six-level computer model.

Appendix C
Semantics Continued

```
Returner ::= return [[ Name | Literal ]]
```

The `Returner` denotes the return value of a function. The keyword `return` shall be followed by nothing or a single `Name` or `Literal`. `Returner` is evaluated as follows.

- `Name` is evaluated to yield a variable identifier. The `Name` is bound to a function declaration. The type of the `Name` has to be the same as the type of the function declaration.
- `Literal` is evaluated to yield a value. The `Literal` is bound to a function declaration. The type of the `Literal` needs to be the same as the type of the function declaration.

```
Single-boolean ::=  Factor (( == | > | < | != | <= | >= )) Factor
    | true
    | false
```

The `Single-boolean` rule in Imp gives the possibility to compare two `Factors`. A comparison shall work with `Float-literal` and `Int-Literal` in `Factor` and will return either `1` or `0`.

- The expression $SB = F_1 O F_2$ is executed as follows. F_1 is evaluated to yield a value, then an operator (`O`) is expected and finally F_2 is evaluated to yield a value. The following describe how the operators are executed.
  - ==: If the value of F_1 is the same as the value of F_2 then the `Single-boolean` is updated with value 1. If they are not the same then the value will be 0.
  - >: If the value of F_1 is greater than the value of F_2 then the `Single-boolean` is updated with value 1. If F_1 is less than the value of F_2 then the value will be 0.
  - <: If the value of F_1 is less than the value of F_2 then the `Single-boolean` is updated with value 1. If F_1 is greater than the value of F_2 then the value will be 0.
  - ! =: If the value of F_1 is different than the value of F_2 then the `Single-boolean` is updated with value 1. If they are the same then the value will be 0.

  - <=: If the value of F_1 is the same or less than the value of F_2 then the `Single-boolean` is updated with value 1. If that is not the case then the value will be 0.
  - >=: If the value of F_1 is the same or greater than the value of F_2 then the `Single-boolean` is updated with value 1. If that is not the case then the value will be 0.

- "`true`" is evaluated as `1`.
- "`false`" is evaluated as `0`.

```
Boolean-expression  ::= Single-boolean
    | (( Single-boolean  (( (( '||' | && )) Boolean-expression )) ))
```

`Boolean-expression` makes it possible to use logical expressions. It consists of either a `Single-boolean` or a `Single-boolean` and a `Boolean-expression` separated by either a && (and) or a || (or). They are executed as follows.

- `Single-boolean` is evaluated to yield "0" or "1".
- $BE = B_1 \&\& B_2$: If B_1 and B_2 both evaluate to either "0" or "1" then this expression will evaluate to "0" or "1" respectively. If they differ, then the expression will evaluate to "0".
- $BE = B_1 || B_2$: If either B_1 or B_2 evaluate to "1" then this expression will evaluate to "1". If neither of them evaluate to "1" then the expression will evaluate to "0".

```
Expression  ::= Term
    |    Term (( (( + | - )) Expression
```

To add or subtract, the `Expression` rule is used in Imp. It consists of either a `Term` or a `Term` and an `Expression` with a plus operator (`+`) or minus operator (`-`) in between. `Expressions` are executed as follows.

- A `Term` is evaluated to yield a value.
- $Ex = T + E$: `T` and `E` are evaluated to values of the same type. Then they are added. It is possible to add a float and integer, which will result in a float.
- $Ex = T - E$: Same as above but instead the values are subtracted. It is possible to minus a float and integer, which will result in a float.

```
Term ::= Factor (( (( '*' | / )) Factor ))*
```

The `Term` is either a single `Factor` or one or several multiplications or divisions with another `Factor`. `Terms` are evaluated as follows.

- $Term = F$: `F` is evaluated to yield a value.
- $Term = F_1 * F_2$: F_1 and F_2 are evaluated to values. Then they are multiplied. It is possible to multiply a float and integer, which will result in a float.

- $Term = F_1/F_2$: Same as above but instead the values are divided. If F_2 is 0, the result is undefined. It is possible to divide a float and integer, which will result in a float.

```
Factor ::= ( Expression )
      |    Name
      |    Float-literal
      |    Int-literal
      |    [Expression]Name
```

A Factor in Imp is used in expressions and is a lower level than Term to meet the precedence rules. It consists of either an Expression between parenthesis, a Name, a Float-literal, an Int-literal or an Expression in square brackets followed by a Name. Factors are evaluated as follows.

- $F = (Expression)$: Expression is evaluated to a value.
- $F = Name$: Name yields the value it identifies.
- $F = Float - literal$: Float is evaluated to a floating point number.
- $F = Int - literal$: Int is evaluated to an integer.
- $F = [Expression]Name$: Yields the value of a given index of an array. Name evaluates to an array. Expression evaluates to an integer denoting the index of the array identified by Name. If the array is out of bounds, the result is undefined.

```
Type-denoter ::=    Identifier
```

To denote types of declarations, the rule Type-denoter is used. This rule contains a single Identifier. Type-denoter is evaluated as follows.

- Identifier is evaluated to either int, string, float, char or void. These are the only types in Imp.

```
FName ::= CapLetter Identifier
```

The FName rule denotes a function in Imp. A function starts with a CapLetter and is followed by an Identifier.

- Every character in a function name is significant. The case of FName is also significant and the first letter must always be a capital letter.

```
Name ::= SmallLetter Identifier
```

The Name rule denotes a variable. Variables in Imp starts with a SmallLetter and an Identifier.

- "Name = SmallLetter Identifier:" evaluates to a variable name starting with a non-capital letter. Every character in Name is significant.

```
Identifier ::=  Letter (( Letter | Digit ))*
```

An `Identifier` starts with a `Letter` and is followed by zero or more `Letter` or `Digit`.

- Every character in an `Identifier` is significant. The `Letter` is evaluated. Then either none or many `Letter` or `Digit` are evaluated.

```
Literal ::=      Integer-literal
    |   Float-literal
    |   String-literal
    |   Char-literal
```

A `Literal` in Imp consists of either an `Integer-literal`, `Float-literal`, `String-literal`, `Char-literal`. `Literal` evaluates in the following manner.

- `Integer-literal` evaluates to an integer
- `Float-literal` evaluates to a floating point number
- `String-literal` evaluates to a string of `Graphic`
- `Char-literal` evaluates to a single character.

```
Integer-literal ::= Digit | Digit Integer-literal
```

The `Integer-literal` is the integers in Imp and consist of a single or several digits. `Integer-literal`s are evaluated as follows.

- `Integer-literal` is evaluated as either one or a series of digits unified into one integer.

```
Float-literal ::= Integer-literal.Integer-literal
```

The `Float-literal` is the floats in Imp and consist of two `Integer-literal` with a punctuation in between denoting a decimal separator.

- `Float-literal` is evaluated as two `Integer-literal`s seperated by a "."

```
String-literal ::=  " Graphic* "
```

The `String-literal` is the strings (text) in Imp and consists of zero or many times `Graphic` between a " and ".

- The value of `String-literal` is given by evaluating zero or multiple `Graphic`.

```
Char-literal ::= '((Graphic))'
```

The `Char-literal` is the char type in Imp, which includes a single `Graphic` between a ' and '.

- The `Char-literal` is given by evaluating `Graphic`.

```
Comment ::= // Graphic* eol
```

A comment in Imp is made by the `Comment`, where // is meant to start the comment, with graphics (text) in between an end of line.

- The "`Comment`" is evaluated as follows. Two // are checked; then zero or multiple `Graphic` is evaluated.

```
Letter ::=  SmallLetter
|   CapLetter
```

A `Letter` is either a `SmallLetter` or a `CapLetter`, which gives Imp the possibility to make use of letters.

- The "`SmallLetter`" yields the value of `SmallLetter`.
- The "`CapLetter`" yields the value of `CapLetter`.

```
CapLetter ::= 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
```

The `CapLetter` production rules contains all the capital letters of Imp.

- `CapLetter` evaluates to its captital letter value.

```
SmallLetter ::= 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
```

The `SmallLetter` production rules contains all the small letters.

- `SmallLetter` evaluates to its small letter value.

```
Digit ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The `Digit` contains all numerals from 0 to 9.

- The value of `Digit` is evaluated to a numeral.

```
Graphic ::= Letter | Digit | space | tab | . | : | ; | , | ~ | ( | ) |
    [ | ] | { | } | _ | ! | ' | ` | '' | # | $ | %
```

`Graphic` includes a `Letter`, `Digit` or a symbol in Imp.

- The value of `Graphics` is the evaluation of `Letter`, `Digit` or any other symbol given in the syntax.

Appendix D
Imp Manual

This document describes how to write programs in the Imp programming language and use ImpT. The first four sections cover variables, expressions, control structures and functions. Section D.5 describes how to use the translator on Imp programs. To get an idea of the capabilities of Imp, last section is a sample program that calculates prime numbers.

D.1 Variables

A variable is declared by a type and a name (line 1), optionally a value can be assigned in the same statement (line 3). Strings are contained within enclosing quotation marks (line 3). Arrays are declared by the keyword `array`, the data type for the contents of the array, the size of the array enclosed by square brackets, and finally the array name. Assigning a value to a specific array index is done on line 6, note that the index value may be the result of an expression.

Listing D.1: Variable declaration and assignment

```
1 int foo = 3;
2 string str = ”foo string”;
3 array float [10]arr;
4 [foo + 2]arr = 32.224;
```

D.2 Expressions

Expressions in Imp are very similar to those of C and Java. Listing D.2 gives an example on how an expression may look like in Imp. The expression is evaluated with precedence to parenthesis, meaning that the $*$ (multiply) and `-` (minus) operators are evaluated before the `/` operator.

Listing D.2: Expressions

```
int foo = 3;
foo = ((foo + 7) * 10) / (foo − 1);
Printi(foo);
```

D.3 Control Structures

Listing D.3 displays how the three different control structures in Imp are constructed. Lines 1 to 5 shows the `if-else` structure. The `else` clause is optional. Lines 7 to 9 is a `while` loop that continues as long as the statement inside the parenthesis evaluates to 1. The last control structure in Imp is the `foreach` loop that iterates over the contents of an array. `foreach` declares a variable in every iteration that is accessible within the loop body. Lines 11 to 13 would print the contents of the `arr` array.

Listing D.3: Control structures in Imp

```
 1  if (foo == 2) {
 2      foo = foo * 2;
 3  }
 4  else {
 5      foo = foo / 2;
 6  }
 7
 8  while (foo >= 20) {
 9      foo = foo - 3;
10  }
11
12  foreach (int t in arr) {
13      Printi(t);
14  }
```

D.4 Functions

An Imp program must have a main function declared to be correct. This function needs to be spelled with a starting capitol letter. Several functions may be declared, but never inside another function declaration. Listing D.4 illustrates a function call with a corresponding function declaration.

Listing D.4: Functions in Imp

```
void Prut (float f) {
    f = Add(12.31, 0.234);
    Printf(f);
}

float Add(float x, float y) {
    float z = x + y;
    return z;
}
```

D.5 Using The Translator

The ImpT (Imp Translator) translates Imp code to Java code.

Contents of the Source Code Folder

`doc`: this folder contains the ImpT documentation generated in Doxygen.
`src`: this folder contains the source code of ImpT.

Running ImpT

To run ImpT you need to have Java installed. The translator is executed by running the command:

```
java ImpT [flags] [input file]
```

where flags are,

default If no arguments are supplied

-c Invokes: Contextual analysis.

-a Invokes: Printing of the AST to a file.

-ca Invokes: Contextual analysis and prints the AST to a file.

-cd Invokes: Contextual and dependence analysis.

-pp Invokes: The pretty printer.

-ccga Invokes: contextual analysis, code-generation and prints the AST to a file.

-cdcga Invokes: contextual analysis, dependence analysis, code-generation and prints the AST to a file.

-bug Invokes: Debug messages to the console. Can be used with all the above arguments.

D.6 Sample Program

Listing D.5: primes.imp

```
void Main (int startPrime, int limitInput) {
    int limit = limitInput;
    array int [limit]potentialPrimes;
    int counter = startPrime;
    foreach (int i in potentialPrimes) {
        i = counter;
        counter = counter + 1;
    }
    foreach (int i in potentialPrimes) {
        int localCounter = 2;
        int isPrime = 1; // 1 = true, 0 = false
        while (localCounter < i && isPrime == 1) {
            int modulus = 0;
            modulus = Modulus(i, localCounter);
            if (modulus == 0) {
                isPrime = 0;
            }
            localCounter = localCounter + 1;
        }
        if (isPrime == 1) {
            Print("Prime detected");
            Printi(i);
        }
    }
}

int Modulus (int a, int b) {
    while (a >= b) {
        a = a - b;
    }

    return a;
}
```

D.6.1 Potential output

Listing D.6: primes.imp output

```
...
Prime detected
10343
Prime detected
10369
Prime detected
10357
Prime detected
10429
Prime detected
10433
Prime detected
10427
...
```

D.6.2 mySmallProgram.imp

Listing D.7: mySmallProgram.imp

```
void Main () {
    float pidecimal = 3.1415;
}
```