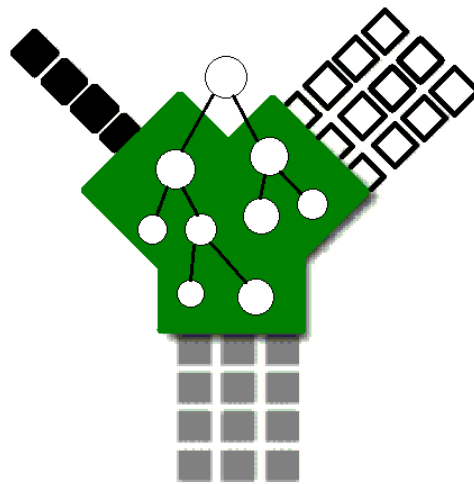

CIP: The Programming Language

A general purpose programming language with SIMD capabilities



**The Faculty of Engineering and Science
Computer Science 4rd term**

Address: Selma Lagerlöfs Vej 300
9220 Aalborg Øst

Phone no.: 99 40 99 40

Fax no.: 99 40 97 98

Homepage: <http://www.cs.aau.dk>

Abstract:

This report is built upon the idea of creating a language that is capable of utilizing SIMD instructions. In this report we will analyse the problems and possibilities of designing such a language and creating a compiler that can compile this language, as well as design decisions that must be taken into consideration while creating our language. We have chosen to call this language CIP. Throughout this report we will go through all the steps necessary to create a general purpose programming language with data parallel capabilities. A context-free grammar is created for the language, which will in turn be used to create an abstract syntax tree. A series of visitors will be created that will traverse the tree, each accomplishing a different task. The last visitor is the code generation visitor which emits code fragments to a specified output file. We benchmark our compiler against GCC's C compiler in order to see if the theoretical increase in performance due to the use of SIMD instruction also reflects in practice. We manage to implement our language and the benchmark shows that our language is better than or on par with GCC compiling an equivalent C program at GCC's lowest optimization option. However, when GCC compiles with optimisation, the C program is much faster than our language.

Project title:

CIP: The Programming Language

Subject:

Design, Definition and Implementation of Programming Languages

Project periode:

Spring 2012

Group name:

D406F12

Supervisor:

Alexandre David

Group members:

Christoffer Moesgaard
Daniel Hillerström
Daniel Rune Jensen
Eric Vignola Ruder
Kimmo V. Andersen
Mathias Ruggaard Pedersen
Søren Kejser Jensen

Copies: 9**Pages:** 91**Appendices:** 12**Finished:** 25-05-2012

The content of this report is publicly available, publication with source reference is only allowed with authors' permission.

Preface

This report is the result of a semester project for the fourth computer science semester at Aalborg University. The project started on the 1st of February 2012 after a presentation of a series of project proposals and a subsequent formation of project groups. We followed two courses alongside this project that have had an impact on the project. The first course was about compiler construction and language design and was taught by associate professor Bent Thomsen. The second course was taught by associate professor Hans Hüttel and was about how to document the syntax and semantics of a programming language.

The material distributed along with this report is available online at <http://dhil.net/public/edu/aau/d406f12/> which includes the full source code of our compiler and the test programs.

Christoffer Moesgaard

Daniel Hillerström

Daniel Rune Jensen

Eric Vignola Ruder

Kimmo V. Andersen

Mathias Ruggaard Pedersen

Søren Kejser Jensen

Contents

1	Introduction	9
1.1	Initial problem	9
1.2	Single Instruction, Multiple Data	11
1.3	SIMD extensions	12
1.4	Problem statement	14
2	Analysis	15
2.1	Streaming SIMD Extensions	15
2.2	Utilising SSE through GCC	17
2.3	Manual optimisation of code using SSE instructions	20
2.4	Branching	20
2.5	Existing solutions	22
2.6	Compiler construction tools	26
3	Design decisions	29
3.1	General language design decisions	29
3.2	Compiler implementation language	30
3.3	Target language	30
3.4	Parser generator	30
4	Language design	33
4.1	Design philosophy	33
4.2	Operational semantics for CIP	33
4.3	Our language	40
4.4	Contextual rules	42
4.5	Language changes	44
5	Compiler architecture	47
5.1	Design patterns	47
5.2	Compiler model	50
6	Test of compiler	67
6.1	Testing Methods	67
6.2	CIP Benchmark	67
7	Conclusion and discussion	71
7.1	Conclusion	71
7.2	Discussion	72
A	Context-Free Grammar	77
A.1	The grammar	77

B	Cip example programs	79
B.1	Hello world	79
B.2	Cross product	79
B.3	Fibonacci	80
B.4	Euclidean	80
B.5	Pythagorean triplets	81
B.6	Matrix manipulation	82
C	Formal semantics for CIP	85
C.1	Formal Semantics	85
	Bibliography	89

Introduction

In this chapter we will give an introduction to the project and present our problem statement. Before presenting the problem statement we will discuss the initial problem analysis.

In this project we will work with programming language design and Single Instruction, Multiple Data (SIMD) instructions and we will try to design and implement a programming language that uses the facilities of the SIMD instructions.

1.1 Initial problem

Our choice of subject and initial problem is born out of curiosity. We have chosen to study how to use the SIMD technology because of the theoretical performance increase that a regular Single Instruction, Single Data (SISD) program could gain under the right circumstances [1]. We want to learn about the concept of true data parallelism which SIMD is based on. Prior to this project none of us have had any experience with the SIMD technology, therefore we begin this project by asking:

What is the SIMD technology and how may it be employed?

Throughout this chapter we will study the concepts of SIMD to obtain a thorough understanding of the technology's capabilities and limitations.

1.1.1 Motivation for data parallelism

To help understand the concept of data parallelism and to show the motivation for using it we have constructed an example which we will refer to throughout this report. The code example in source code 1.1 is written in a high-level C-like syntax, we use a mathematical set notation to denote the contents of the vectors A and B .

```

1  // Vector containing n even numbers
2  int A[n] = { 2k |  $\forall k \in \mathbb{N}, k < n$  };
3  // Vector containing n odd numbers
4  int B[n] = { 2k+1 |  $\forall k \in \mathbb{N}, k < n$  };
5
6  // B + A
❶ 7  for (int i = 0; i < n; i++)
8      B[i] = B[i] + A[i];
9  // Square A
❷ 10 for (int i = 0; i < n; i++)
11     A[i] = A[i] * A[i];

```

Source code 1.1: Additive and multiplicative arithmetic laws of object parity.

The simple code example in source code 1.1 demonstrates two of the mathematical laws of vector arithmetic, namely the addition ❶ and multiplication ❷ of vectors. On their own these laws of vector arithmetic are not of interest to us, however in source code 1.2 we have compiled ❶ into x86-64 assembly code, using GNU Compiler Collection (GCC). Instructions not involved in the actual mathematical operation have been omitted.

```

1  jmp .L2
2 .L3:
3 //Adds B[i] and A[i] and assigns the result to B[i]
4 movl -4(%rbp), %eax
5 cltq
❶ 6 movl -96(%rbp,%rax,4), %edx
7 movl -4(%rbp), %eax
8 cltq
❷ 9 movl -48(%rbp,%rax,4), %eax
10 addl %eax, %edx
11 movl -4(%rbp), %eax
12 cltq
❸ 13 movl %edx, -96(%rbp,%rax,4)
14 //Increments i by one
15 addl $1, -4(%rbp)
16 .L2:
17 //Compares i with n to check if the loop should terminate
18 movl -4(%rbp), %eax
19 cmpl -8(%rbp), %eax
20 jl .L3

```

Source code 1.2: Vector addition in assembly language.

A quick analysis of the code reveals that the add-instruction is applied sequentially one element at a time. ❶ reads the value $[B + 4i]$ from memory and then places it in the 32-bit register **EDX**, and $[A + 4i]$ is loaded into **EAX** at ❷. In ❸ the contents of **EDX** and **EAX** are added and the sum is stored in $[B + 4i]$. This procedure is repeated n times. From this we can see that the same instruction is being applied on a single piece of data of an arbitrary size n times. We see the same pattern in the example code in source code 1.3 which shows the x86-64 assembly code for squaring a vector.

```

1  jmp .L2
2 .L3:
3 //Squares A[i] and assigns the result to A[i]
4 movl -4(%rbp), %eax
5 cltq
6 movl -48(%rbp,%rax,4), %edx
7 movl -4(%rbp), %eax
8 cltq
9 movl -48(%rbp,%rax,4), %eax
❶ 10 imull %eax, %edx
11 movl -4(%rbp), %eax
12 cltq
13 movl %edx, -48(%rbp,%rax,4)
14 //Increments i by one
15 addl $1, -4(%rbp)
16 .L2:
17 //Compares i with n to check if the loop should terminate
18 movl -4(%rbp), %eax
19 cmpl -8(%rbp), %eax

```

20 j1 .L3

Source code 1.3: Vector squaring in assembly language.

Again we see a very similar pattern, where the instruction is being applied successively at ❶. We will now consider the possibility of applying the instruction on multiple data in parallel.

1.2 Single Instruction, Multiple Data

SIMD is a special instruction set which allows a single operation to be applied to multiple data and is supported by most modern processors [1]. Both SIMD and SISD are part of Flynn's taxonomy together with MISD and MIMD which means Multiple Instruction, Single Data and Multiple Instruction, Multiple Data, respectively. Flynn's taxonomy can be represented visually as seen in figure F1-1 which describes all four computer architectures, with the first dimension representing the amount of instruction streams that a computer architecture can process at a time and the second dimension representing the amount of data streams that a computer architecture can process at a time. [2] In this project we will not be concerned with multiple instruction architectures.

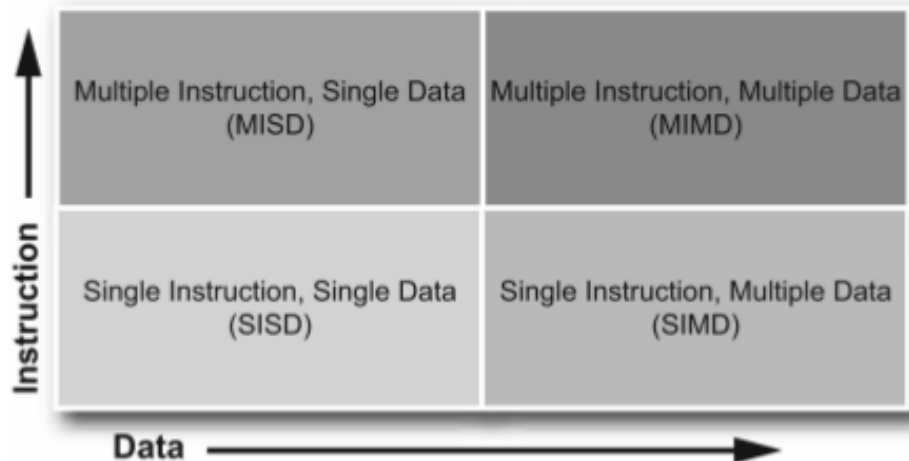


Figure F1-1: A visual representation of Flynn's taxonomy [2].

SIMD differs from SISD in the amount of data that can be manipulated per instruction. In SISD the CPU can only send a single instruction stream to a single stream of data stored in memory each clock cycle. SIMD is capable of sending a single instruction stream to multiple data streams at once thus applying the instruction on the data streams in parallel. [2] This concept can be seen in figure F1-2 where SISD takes a single instruction that is used on a single piece of data, giving a single result. SIMD on the other hand takes a single instruction and uses it on multiple pieces of data, which gives a number of results equal to the amount of input data.

SIMD is therefore beneficial in applications of multimedia which usually have large amount of data with consistent associations. The data processing capabilities of SIMD effectively increase the efficiency of applications with a large amount of data with consistent associations compared to similar application that only utilise SISD technology. [4] Furthermore, when using

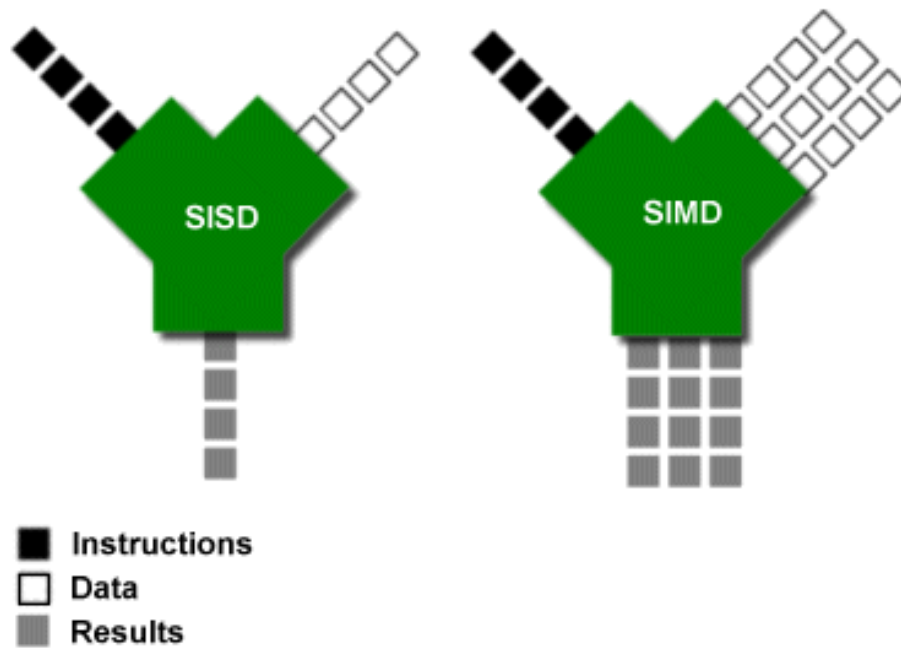


Figure F1-2: The difference between SISD and SIMD [3].

SIMD, power consumption is reduced in most appliances compared to using standard SISD. [5]

Some modern compilers support the utilisation of SIMD technologies through optimisation routines. However it is not possible for the programmer to specify where to use the SIMD instruction and as such the programmer can not know which parts of the program are optimised. Therefore it may be necessary to write SIMD subprograms in assembly language rather than having the compiler construct the appropriate SIMD program in order to gain control of when the technology is used. [1] Some compilers allow compiler-specific language extensions that allow more control over how SIMD is used. For example, GCC has an extension which provides a way to define arrays as vectors, meaning that operations on these vectors will be performed using SIMD instructions. These extensions are described in section 2.2.3.

1.3 SIMD extensions

SIMD is supported by most CPUs today and is included in the instruction set of the CPU, which can be studied through the manuals published by the industry and implemented directly as assembly code.

SIMD describes any extension to CPUs that allows the CPU to execute instructions on data in parallel. Some of the most common SIMD extensions to CPUs are MMX, SSE, and 3DNow! and most of these sets have been extended further since their first release. These extensions can be seen in figure F1-3, which shows a timeline of the main SIMD extensions. As can be seen from figure F1-3, MMX was the first SIMD extension and others have since been developed, some of which have extended the MMX instruction set. [6]

The Intel branch of the SIMD extensions will be the focus of this project.

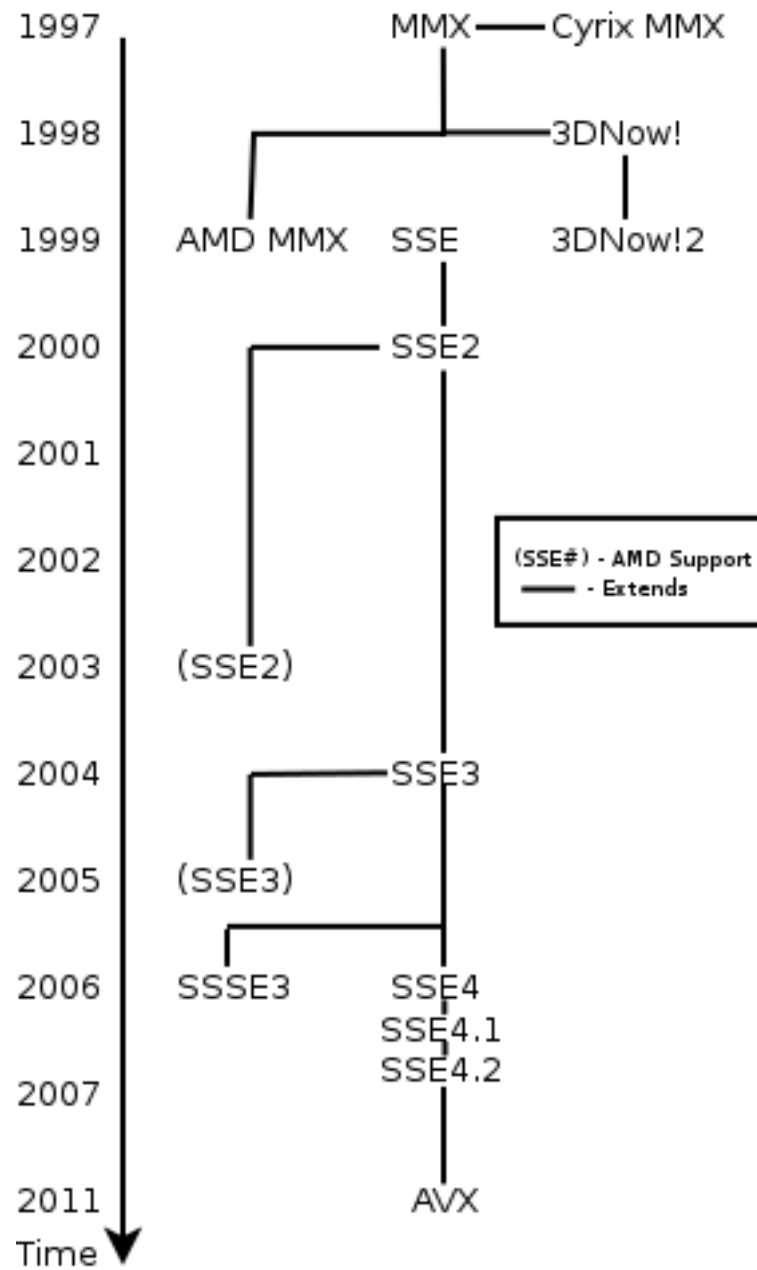


Figure F1-3: SIMD extension timeline.

The *MMX* extension introduced eight new 64-bit general purpose registers which could be used in a number of ways by its instructions, but occupied the same register space as the *floating-point* registers, thus not being able to use these at the same time. Since then the MMX extension has been extended further into 3DNow!, which is an AMD extension of the MMX. Intel extended the MMX with Streaming SIMD Extensions (SSE) that added new 128-bit registers in another, separate register space, thus making it possible to use floating-point and SIMD at the same time. After this the SIMD extensions *SSE2*, *SSE3*, *SSSE3*, *SSE4.1*, *SSE4.2*, and *SSE4a* did not expand the registers but only added additional instruction sets to perform more delicate operations on the registers. [6]

1.4 Problem statement

SIMD has proven faster when it is possible to structure the computations as parallel computations, for example in the Xvid video encoder [7] which has used SIMD instructions instead of SISD in order to improve performance. We want to know if it is possible to create a programming language and a corresponding compiler, that allows the programmer to seamlessly use SIMD in conjunction with SISD, with little differentiation between the two technologies, thereby allowing the programmer to focus on the problem at hand instead of the implementation. At the same time, the language should not sacrifice the performance gained through SIMD since SISD would then be a more preferable solution, as it does not require specific data structures like SIMD. This has led us to the following problem statement.

How may we design and implement a programming language that utilises the concepts of SIMD?

Our problem statement resulted in the following questions, which together with the problem statement is the foundation of the design, implementation and testing of our programming language documented in the rest of this report.

- How can such a language be formalised?
- Which data types should be established?
- How can we design constructs that are easy for the compiler to generate SIMD instructions from?
- How can we encourage the programmer to use these constructs?
- How does the performance of our language compare with already established compilers?

Analysis

In this chapter we will look at problems related to SIMD parallelism and solutions to these problems, most importantly the problem of branching, and different implementation methods for SIMD. We will also look at other parallel programming languages in order to gain inspiration for our own language.

2.1 Streaming SIMD Extensions

SSE was introduced in 1999 and is like MMX an extension to the Intel and AMD microprocessors. It was originally released as an extension for the Intel Pentium III and AMD AthlonXP series. SSE differs from MMX by adding a new, separate register space to the microprocessor. Due to this the usage of SSE requires the operating system to support it. However, SSE has been supported on Windows and Linux since Windows 98 and Linux kernels 2.2. [6]

SSE added eight new 128-bit registers **XMM0** through **XMM7**, collectively called **XMMi**. These registers can store the new *packed* data type introduced in SSE, which consists of four 32-bit single-precision floating point values [6]. Beside the eight new registers an additional register **MXCSR** was also added. The **MXCSR**-register is a 32-bit control register which contains flags for control and status regarding SSE instructions [6]. The 32-bit version of the entire SSE programming environment can be seen in figure F2-1. In a 64-bit version, the amount of registers would double to 16, and the address space would be increased to $2^{64} - 1$.

In addition, SSE added 70 new instructions that operate on the new 128-bit registers, the MMX registers, and some that operate on the regular 32-bit registers [6]. SSE contains several interesting instructions such as floating point arithmetic instructions for addition, subtraction, multiplication, and division on multiple pieces of data. These are called **ADDPS**, **SUBPS**, **MULPS**, and **DIVPS**, respectively, and are capable of performing basic arithmetic operations on four single-precision values with four other single-precision values. A visual representation of how these operations work can be seen in figure F2-2. Similar instructions for other data types are similar except for the amount of data elements that can be worked on at a time due to the size of the registers.

The compare instruction **CMPEQPS** is capable of comparing two packed data types, containing four single-precision values each, to determine whether each value is equal to the corresponding value in the second packed data type. This particular instruction has several sibling instructions, one for each of the relational operators ($=, \leq, <, \geq, >$ etc.).

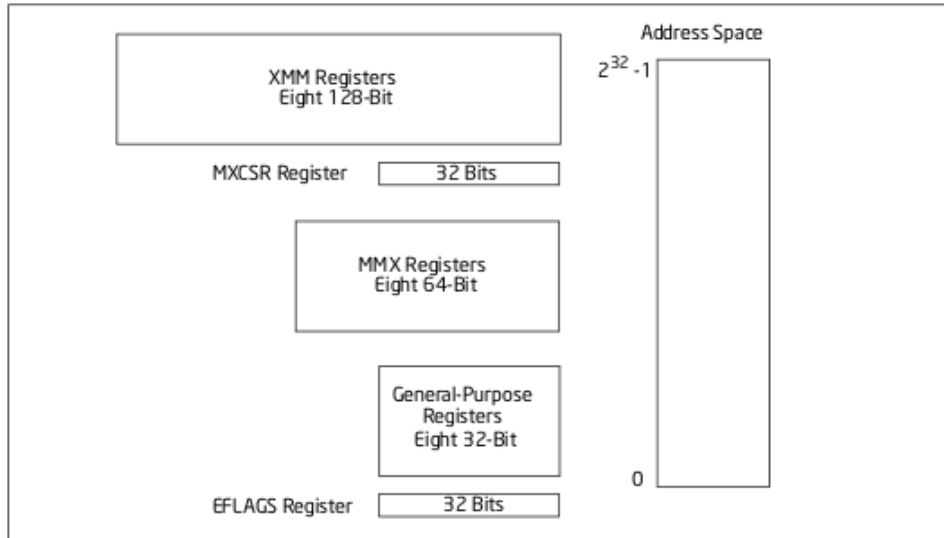


Figure F2-1: SSE programming environment for x86 [8].

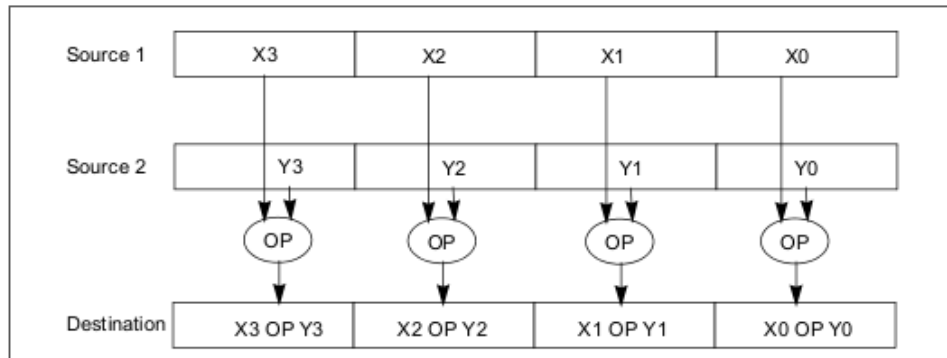


Figure F2-2: Arithmetic operations on four data elements in parallel [8].

2.1.1 Streaming SIMD Extensions 2

Streaming SIMD Extensions 2 (SSE2) was introduced by Intel in 2000 for the Pentium 4 microprocessor, and adopted by AMD in 2003 for their Opteron and Athlon 64 processors. SSE2 uses the same XMM_i registers as SSE [6] which means that SSE2 does not need any additional operation system support, as long as SSE is supported [8]. The main new feature introduced in SSE2 is five new packed data types as can be seen in figure F2-3. These new packed data types allow integer and double-precision floating point computations to be performed using the XMM_i registers [8].

SSE2 also introduced 144 new instructions in conjunction with the new packed data types. These mostly include modifications of the existing instructions, to accommodate integer and double-precision floating point calculations. Additionally, SSE2 introduced a new interesting instruction for controlling how the processor uses its cache. [6, 8].

2.1.2 Streaming SIMD Extensions 4

Streaming SIMD Extensions 4 (SSE4) was officially announced on the 27th of September, 2006. Both Intel and AMD included the SSE4 technology in their processor releases in early 2007. [6, 9] SSE4 was at the time Intel's

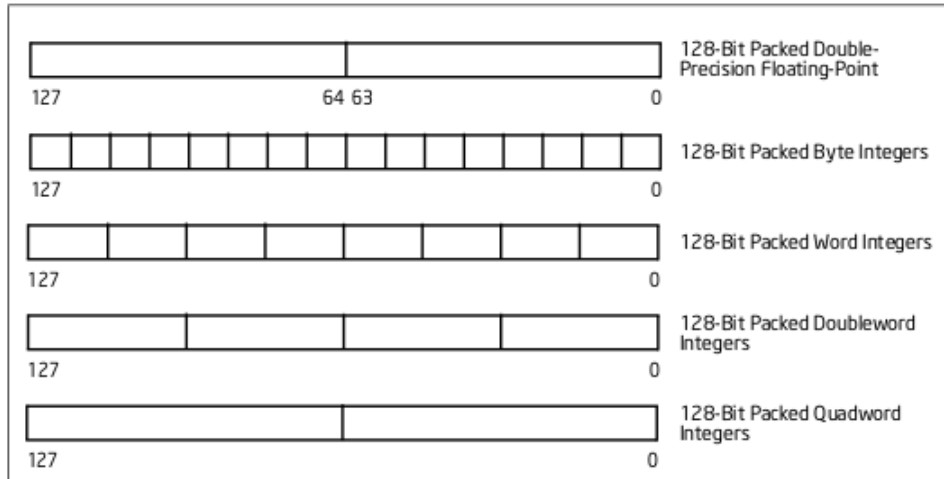


Figure F2-3: New packed data types introduced in SSE2 [8].

largest ISA extension since SSE2 in terms of scope and impact [9]. Among the new extensions were the, as described by Intel, “new and innovative” string processing instructions [9].

SSE4 is available in three flavours:

- SSE4.1
- SSE4.2
- SSE4a

SSE4.1 includes a total of 47 new instructions, SSE4.2 extends this instruction set by including seven new instructions. The SSE4a is an AMD variant and therefore it does not support the full set of SSE4 instructions, however it included six additional instructions for bit manipulation [6].

New instructions allow packed multiplication of four either signed or unsigned 32-bit integers [9]. Such instructions help utilise the capabilities of the SIMD/SSE4 architecture thus effectively improving the throughput rate per clock cycle. New string instructions included in SSE4.2 provide a comprehensive set of string processing capabilities [9]. These instructions were designed to improve applications of string processing commonly found in XML parsing, databases, compilers, etc.

2.2 Utilising SSE through GCC

There exist multiple methods which can be used to utilise the possibilities of SSE2 without writing entire programs directly in assembly language. These methods unfortunately often impose a certain degree of constraints on the programmer, in an effort to make the resulting code easier for the compiler to interpret and translate into SIMD instructions. The following is a description of three such methods that use the C programming language and GCC. Our reasons for using the C programming language and GCC as a target platform are documented in chapter 3.

2.2.1 Inline ASM

The C programming language allows the programmer to write assembly language inline in the C program through the `ASM` function. This allows the programmer to write assembly language and use SSE2 or other parts of the instruction set with the rest of program implanted in higher level C code. The GCC C compiler also supports an extended version of the `ASM` function which removes the need to manage registers directly, and makes it easier to interface with C variables. The code example in source code 2.1 demonstrates GCC's extended version of the `ASM` function.

```
1 __asm__(  
2     "Assembly instructions"  
3     : Output Variables  
4     : Input Variables  
5     : Clobbered List  
6     );
```

Source code 2.1: GCC Extended inline assembly.

The extended `ASM` function is composed of three parts: The first is a string of assembly language instructions, the second is a list of input and output variables and how GCC should manage them, and the third is a list of elements clobbered by the execution of the assembly instructions. The input and output lists allow the use of external variables in assembly without the need to think about register allocation. The clobbered list is needed to inform GCC of what the assembly instructions change, for example the contents of a register or memory locations. This is to prevent optimisation that might break the code or cause corruption of the results [10].

Using SSE2 through the extended `ASM` function allows the programmer complete control over which instructions to use, and how the data should be modified, and it is easily interfaced with the surrounding code through variables. Another important aspect of the `ASM` function is the possibility to use the entire CPU instruction set, including instructions that are not part of the SSE2 instruction set or the main programming language used. The downside with `ASM` is that it requires knowledge of the assembly language used by the target platform and its CPU architecture.

2.2.2 Vectorisation

Vectorisation means transforming the data and operations of the program in question from a *scalar* form to a *vector* form [11]. The difference between a scalar and a vector operation is that a scalar operation only works on a single piece of data at a time in a sequential fashion, whereas a vector operation works on two or more pieces of data at a time in parallel.

Usually the most important construct in a program as a candidate for vectorisation is the loop construct [11]. In order to vectorise a loop, one must unroll the loop with an unroll length corresponding to the length L of the vector, and insert vector instructions instead of scalar instructions. Thus, for each iteration of the loop, the processor can make L computations in parallel instead of a single computation.

There are restrictions on which parts of a program that may be vectorised, however. Most importantly, there must be no dependencies between the pieces of data that are computed in parallel. That is, if four pieces of

data a, b, c, d are computed in parallel, but the computation of c depends on the result of a , then these four computations will not be able to be computed in parallel.

GCC has automatic vectorisation capabilities that allow it to vectorise some patterns of code [12]. These include support for computations of arrays, both single-dimensional and multi-dimensional.

Using GCC's vectorisation capabilities is in most cases enough to vectorize loops or small blocks of code, but there are certain loops it does not yet support, mainly loops without a clear end condition. In source code 2.2 for example, GCC is not capable of vectorising the loop because the end condition can not be computed beforehand [12].

```
1 while (*p != NULL){
2   *q++ = *p++;
3 }
```

Source code 2.2: Loop without specific end condition. [10]

These requirements force the programmer to actively format the computations in a way that is suitable for GCC to vectorise. Doing this requires knowledge about how GCC optimises the source code, and how the vectorisation routines work, in addition to eliminating data dependency, and the only way to determine if the result is as expected is to read the generated assembly code.

2.2.3 GCC vector extensions

Another way of using SIMD through GCC is through its vector extensions. These extensions make it possible to operate on vectors of data, defined using the `typedef` keyword, in the same way as simple data types. An example of this is shown in source code 2.3.

```
❶ 1 typedef int v4si __attribute__((vector_size (16)));
   2
   3 v4si vector_a, vector_b, vector_c;
   4
   ❷ 5 vector_c = vector_a + vector_b;
   6
   ❸ 7 vector_c = vector_a + 1;
```

Source code 2.3: GCC vector extensions. [10]

A vector is defined at ❶. The size of the vector is defined as sixteen bytes, which is divided into a number of integers. If the implementation uses four bytes for the `int` data type, then the sixteen byte vector, would be divided into four units, each with a size of four bytes. This new type is then used to instantiate variables which can be used like simple data types. This is shown at ❷ where two vectors are added together. The compiler translates the addition into suitable packed data types and SSE instructions, and the result is then assigned to `vector_c`. The vector data types can also be used together with simple data types. At ❸, for example, a scalar is added to `vector_a`. This scalar is expanded into a vector corresponding to `vector_a` before they are added.

This extension is simple to use, since it lets GCC handle the SIMD implementation through code generation. The extension unfortunately only supports simple mathematical and binary operators at the moment. The

same problem which was present when utilising GCC's vectorisation is also present here, which is that it is not possible to ensure that the SIMD code generated by GCC is as intended without reading through the assembly language code generated [10].

2.3 Manual optimisation of code using SSE instructions

Returning to the code examples from section 1.1.1 we will apply a simple and small optimisation using the SSE instruction set. In source code 2.4 we show an optimised C inline x86-64 assembly code implementation of the vector addition in source code 1.2 using the SSE instruction set.

```

1 void vectorAdd_SIMD(int n, int src[n], int dest[n]) {
2     __asm__(
3         // rsi is the loop counter and is set to zero
4         "xor %%rsi,%%rsi\n\t"
5         // Let rcx and rdx point to A and B respectively
6         "movq %[A],%%rcx\n\t"
7         "movq %[B],%%rdx\n\t"
8         ".additionloop:\n\t"
9         // Load [A+4i] into xmm0
10        ❶ "movups (%%rcx,%%rsi,4),%%xmm0\n\t"
11        // Add [B+4i] to xmm0
12        ❷ "addps (%%rdx,%%rsi,4),%%xmm0\n\t"
13        // Move result into [B+4i]
14        "movups %%xmm0, (%%rdx,%%rsi,4)\n\t"
15        // Determine whether to continue loop
16        ❸ "addq $4,%%rsi\n\t"
17        "cmp %[n],%%esi\n\t"
18        "jle .additionloop\n\t"
19        : "=m" (*dest)
20        : [n] "r" (n), [A] "m" (src), [B] "m" (dest)
21        : "cc", "%rsi", "%rcx", "%rdx", "%xmm0", "memory"
22    );
23 }

```

Source code 2.4: SSE variant of code example 1.2.

In the code example in source code 2.4 it is assumed that n is a multiple of 4. This particular implementation unrolls the arrays by 4 ❸ to ensure that the XMM0-register is properly filled ❶ during each iteration. The ADDPS ❷ effectively adds four 32-bit integers, stored from $[B + 4i]$ and 128 bits ahead, to the four 32-bit integers within XMM0. Thereby we see that we apply a single instruction to multiple data.

2.4 Branching

Modern microprocessors are designed with pipelines to enhance their performance. Pipelines allow the processor to fetch and decode the next set of instructions while another is being executed. A problem when using a pipelined processors is branching. If there is only one pipeline in the processor then it has to predict which branch to fetch instructions from. If the processor mispredicts which way the branch goes, then the pipeline has to be flushed and the instructions from the correct branch have to be fetched.

This is costly since the cycles used when filling the pipeline were wasted [13, 14].

Branching is even more expensive when using SIMD, since branching does not exhibit parallel behaviour, as the comparison of elements from multiple packed data type could give different results. Implementing normal compare/jump branching with packed data types would then require each element to be computed individually, with a substantial performance decrease as a result, in addition to the possibility of branch misprediction [15].

2.4.1 Masking

Masking is a way of circumventing the problems of branching, while still maintaining the performance benefits gained through SIMD [16]. A mask is essentially a crafted sequence of bits, which is used to alter data elements with the use of bitwise operations. Using masking solves the problem with parallelism and the pipeline. Masking does not make use of the pipeline, and there is therefore no chance of a mispredict and subsequent flushing of the pipeline. Since masking is done with simple operations, it also does not impose any significant performance loss.

Utilising masking instead of branching allows us to compare multiple equally sized packed data types, value by value in a single instruction, with a mask as a result. The bit sequence of the mask is the result of the comparison, where bits corresponding to elements which passed the comparison are set to one, and bits corresponding to elements that failed the comparison are set to zero. The mask can then be used together with bitwise instructions to include or exclude elements in the packed data type from computations, by setting them to zero.

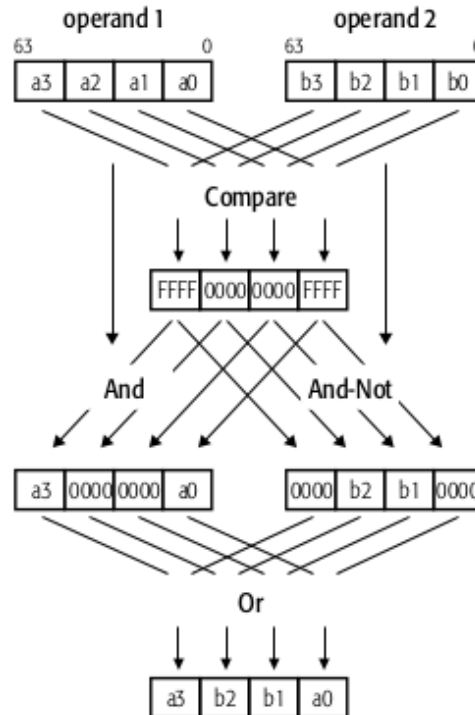


Figure F2-4: Masking in practice. [15]

Figure F2-4 illustrates the use of masking to combine two packed data types according to a comparison. Here the first and last elements of the packed data type are true according to the comparison, so the entire bit sequences of these elements are set to ones, while the comparison of the two middle elements was determined to be false and their bits are set to zero.

A bitwise AND is then used to make a new packed data value with the elements from packed data type a that passed the condition, and a bitwise AND NOT to choose the elements from b that passed the condition. A bitwise AND NOT is used since the mask created corresponds to the elements in a , so the elements in b that passed the condition must correspond to those that failed in a . The bit mask could also be used on other packed data values than the ones used to create it, if for example we only wanted to add values to the elements that passed a condition, or choose elements in one packed value based on the what values another one contained.

Using masking allows us to determine computations dynamically at run-time, without the drawbacks of branching, and is in many cases a necessity in order to utilise SIMD without resorting to sequential scalar computations. [17, 16]

2.5 Existing solutions

In this section we will look at existing parallel programming languages in order to gain some knowledge and inspiration from them. This is done to get a better understanding of the focus of this report and get inspiration as to how we would like the language of this project to be. We have found a collection of languages that utilise SIMD in a variety of ways. Key parallel language features are going to be described and analysed. Before these parallel languages are represented, terminology about different forms of parallelism in software development is explained to specify the routes a parallel programming language can take.

2.5.1 Terminology concerning parallelism

On the topic of parallelism, we distinguish between two forms of parallelism.

Data parallelism is a form of parallelism in which the same operations are applied to different data items at the same time. The amount of parallelism is thus scalable, since it can be expanded to include more units of data.

Task parallelism on the other hand works by performing a number of tasks in parallel. Task parallelism does not scale as well as data parallelism, since the only way to increase the amount of parallelism is to divide the program into smaller parts that may be run parallel, which is not always possible.

Thus, data parallelism is concerned with data while task parallelism is concerned with processes. In reality, however, most parallel programs are a hybrid of the two. In this report we have scoped down to focus on SIMD. This means that our focus will mostly be on data parallelism, since that is the kind of parallelism that is supported by SIMD. [18]

In addition, parallelism can be further categorised with regards to who is responsible for declaring and handling the parallelism of a given program [18].

Implicit parallelism means that only the hardware is responsible for making use of parallelism, possibly with the compiler helping to make the code more easily run in parallel.

Explicit parallelism means that the programmer, through constructions in the language, is responsible for declaring where in the code the compiler should make use of parallelism.

In our programming language, we wish to exploit explicit parallelism so that it is easier for the programmer to make use of parallelism while designing the language in such a way that the programmer is encouraged to structure his or her code in a fashion that is easy for the compiler to translate into a format that utilises data parallelism.

2.5.2 Parallel languages

Some programming languages contain features that makes data parallelism easier to use for the programmer through the platform’s SIMD instructions. This is mostly done either by refactoring code when compiling the program, as in implicit parallelism, or by implementing special data types and functions to create an abstraction from the underlying hardware implementation, as in explicit parallelism. The following is a brief explanation of some different approaches to data parallelism that these languages use.

2.5.2.1 NESL

NESL is a programming language that was designed to make parallel programming easy [19], much like the programming language we envision.

The main construct in NESL is the *sequence*, which is very similar to an array in other programming languages. A sequence is defined with its elements between brackets:

[2, 5, 8, 4]

The preceding sequence contains four elements: 2, 5, 8, and 4. One of the main features of NESL is the ability to apply operations to all the elements of a sequence in parallel using an apply-to-each construct. These can be operations such as arithmetic operations or printing strings in reverse [19].

```
1 {a + b : a in [3, -4, -9]; b in [-11, 2, 3] | a > 0 and b
  < 0};
```

Source code 2.5: Addition of values in sequences with conditions in NESL.

An example of NESL code can be seen in source code 2.5. In this example, the elements of the two sequences *a* and *b* are added together in parallel. The arguments after the pipeline give restrictions as to which elements are used. In this example, only values from the sequence *a* that are larger than 0 are used. Likewise, only values that are less than 0 are used from *b*. This means that only 3 from *a* and -11 from *b* are added together, returning a sequence [-8].

2.5.2.2 ZPL

ZPL is an array programming language designed with performance in mind [20]. The main construct in ZPL is the n -dimensional array. One of the main features of ZPL is its *region* property that forces the programmer to specify the dimensions of the arrays that is being worked on in a given part of the program. These regions can additionally be modified throughout a program to work on different parts of a matrix.

In ZPL, the programmer will not have to specify how parallelism is to be performed in the program. Rather, parallelism is automatically provided for many of the constructs of the language when it is compiled. [21] It has two different types of array: One type that does not support parallelism but allows for indexing, and another type that supports parallelism but does not allow indexing. Any operations on arrays that support parallelism are automatically vectorised by the ZPL compiler.

```

1 program main;
2
3 config var
4   n : integer = 2;
5
6 region
7   R1 = [1..n, 1..n];
8   R2 = [1..10, 1..10];
9
10 direction
11   right = [0,1];
12
13 procedure main();
14   var
15     A : [R1] integer;
16     B : [R2] integer;
17     C : [R2] integer;
18
19   [R1] begin
20     A := 1;
21     B := 2;
22
23     C@right := A + B;
24
25     writeln(C);
26   end;

```

Source code 2.6: Addition of arrays with ZPL.

An example of ZPL code can be seen in source code 2.6. In this example, two arrays are added together and the result stored in a third array. In the declaration section at ❶, an integer n , two regions $R1$ and $R2$, and a direction *right* are defined. Note that region $R2$ is larger than $R1$. Within the main procedure at ❷, A is initialised with region $R1$ whereas B and C are initialised with region $R2$. When not initialised with a value, scalars and arrays are set to one by default in ZPL. At ❸, $R1$ is used as the region from then on. This means that any assignments or computations done on arrays within this section will only be done on the part of the arrays in question corresponding to the region. So for this example, only the first 2×2 elements of any array will be used. All of these elements are set to one in A and to two

in B . At ❹, the *right* direction operator pushes all operations one step to the right, as defined in the declaration section. This means that the result of the addition will be stored in the entries (1, 2), (1, 3), (2, 2), (2, 3) (assuming a 1-based matrix).

Thus, when the array C is printed at ❺, and the direction operator is no longer used, the entries (1, 1), (1, 2), (2, 1), (2, 2) are printed, yielding the results 0, 3, 0, 3, respectively.

2.5.2.3 Mono

Mono, the open source version of .NET and C#, lets the programmer control more explicitly when data should be computed in parallel compared to ZPL and NESL. Mono includes vector classes that contains methods for the different SSE instructions. This approach gives the programmer control over how the instruction are utilised, but forces them to learn yet another language construct to use it.

```

1 using System;
2 using Mono.Simd;
3
4 static class MainClass
5 {
6     public static void Main()
7     {
8         Vector4i v1, v2, v3;
9         Vector4f v4, v5;
10
11         v1 = new Vector4i(1, 2, 3, 4);
12         v2 = new Vector4i(4, 3, 2, 1);
13         v3 = new Vector4i(3, 5, 7, 11);
14
15         v1 = v1 + v2;
16         v3 = v2 * v1;
17
18         v4 = new Vector4f(16, 25, 100, 1225);
19         v4 = new Vector4f((float) 89.98, 2, (float) 45.9,
20             (float) 13.17);
21
22         v4 = v4.Sqrt();
23         v5 = v5.CompareLessThan(v4);
24         v4 = v5 & v4;
25
26         System.Console.WriteLine(v4);
27     }
28 }

```

Source code 2.7: Various SIMD operations implemented in Mono.

In order to more easily facilitate SSE operations, C# is expanded with types corresponding to the packed data types defined in the CPU architecture, these are declared at ❶. These can be used to perform normal scalar operation, on the multiple elements contained in the vector as seen at ❷. Mono's implementation also implements some of the more advanced features of SSE as C# extension methods. These are used at ❸ to compute the square root of $v4$.

This makes the instructions more accessible, since the programmer is not forced to remember assembler instructions, and makes the instructions

available through an object oriented interface [22].

2.5.2.4 EXPAND

A third solution to this problem was proposed by Jaewook Shin and described in the paper Programming by Expansion [23]. The paper describes the developed EXPAND compiler, which uses the same language for both input and output, in this case the C programming language. The EXPAND compiler reorganizes the code at the function level, and substitutes scalar operations with vector equivalents, so that it is easier for the real C compiler to create code utilising the SIMD instructions. This should in theory allow the programmer to write code with minimal consideration for SIMD, and then let the EXPAND compiler generate functions capable of using SIMD, which the programmer then can tune to their liking before compiling to machine code.

2.6 Compiler construction tools

When crafting a compiler, it is possible to craft a custom and hand-made lexical analyser as well as a parser, but this is a very time-consuming process, as well as being prone to human error. There are several tools available which facilitate these steps. By using such tools, a lot of the process is automated. This saves time because of the fact that the lexer and parser do not need to be hand-written and debugged. The language is instead simply defined by a set of regular expressions and Backus-Naur derivation rules which the tools use to generate the necessary code. There is a need to create a lexer that recognizes strings to see whether or not all inputs are acceptable in our programming language. Then a parser needs to be generated to see whether or not all the input strings are written syntactically correct. Lastly, an Abstract Syntax Tree (AST) can be created with the help of the parser, to facilitate semantic analysis, code optimisations and code generation. In the following, we will look at a selection of available tools and give a brief description of each.

2.6.1 Tool suites

There are several different approaches to all these different steps. One approach is to have two different tools: One that generates the lexer, another that generates the parser, and lastly there are tools which are capable of both, generating both a lexer and a parser. Depending on which tool suite is used, the AST is created as well. If the AST is not automatically created by the parser, it is often possible to do so by assigning action commands to each production in the grammar. This means that it is possible to run specific commands whenever a production is met, such as creating instances of a class and assigning values to these classes.

A very well-known parser generator suite is called SableCC [24]. This suite produces a LALR parser by defining tokens with regular expressions and then describing the grammar in Extended Backus-Naur Form (ENBF). This is one of the tools that automatically creates an AST. This is done by giving specific names to each possible production in the language's grammar. This way it is possible for the parser to create AST nodes from these names.

Another well-known parser generator suite is called JavaCC [25]. This is similar to SableCC in that it creates a parser and lexer from regular expressions and ENBF to describe the grammar. The main difference though, is that JavaCC is a top-down parser and thus limits the class of applicable grammars to LL(k) grammars. It also does not automatically create an AST, which is done using another tool called JJTree in combination with JavaCC.

The last tool suite we have looked at is a suite called Coco/R [26]. This tool suite is very similar to JavaCC as it creates a top-down parser from an LL grammar. One difference between the two, is that Coco/R is limited to the LL(1) class of languages, and does not create an AST automatically, nor by an external tool. The AST is instead created by writing action code in the grammar itself.

2.6.2 Standalone tools

Another approach to lexer and parser generation is to generate the lexical analyser and the parser independently and then link them together by making the lexical analyser conform to the standards imposed by the parser. Then, action commands would be inserted into the parser in order to generate the AST.

One of the most commonly used lexer generators is one called Lex, which was developed by AT&T [27]. Lex generates a lexer from a set of regular expressions defined by the user. Flex is another lexer generator which stands for Faster Lexical Analyser [28]. Flex generates a lexer in C, but there exists other versions of the program like JFlex, which is a Java version.

After generating a lexer, a parser needs to be generated. Usually the parser is built at the same time as the lexer because the parser is usually capable of creating a symbol table for the tokens that the lexer can use when reading input [29]. This means that the tokens are defined in the parser generator which then creates a symbol table for the lexer to use.

For this job there are two well known parser generators: CUP and Byacc. These tools are both fully compatible with JFlex and Flex, respectively. By telling Flex or JFlex to conform to Byacc's or CUP's standards, no further work is required in order to make both programs work with each other [30]. Both parser generators are actually rather similar, and use a Context-Free Grammar (CFG) written in a BNF-style syntax. They both also use action code to define the construction of the AST [29].

Design decisions

In this section we will define further restrictions in order to scope our product, namely the compiler for our programming language.

As described in section 2.1, there are several extensions to SSE, but all the instructions that we are interested in using are available in SSE2, and since SSE2 is an older extension than SSE4 and thus more widely used, we have to chosen to focus on the SSE2 instruction set. Since we will be focusing on utilising SSE2 instructions to perform computational intensive operations in our compiled programs the target platform will obviously have to support this technology.

We will write our compiler in Java, but the compiler will not produce code that will be interpretable by the Java Virtual Machine (JVM). Instead our compiler will produce C code with inline x86-64 assembly code which should be compilable by GCC. This may seem like a peculiar match, and in the following sections we will account for our choices.

3.1 General language design decisions

Our analyses of existing implementations of data parallelism and specifically SIMD showed that each had both pros and cons associated with them. ZPL, which we described in 2.5.2.2, allows the programmer to just write their programs without thinking about how the results are computed. This makes it very easy to utilise, but very hard to define how specific computations are handled. The approach of Mono that is described in section 2.5.2.3 allows the programmer to specify when to use parallelism, and partially how, which gives the programmer more control over how the instructions are used. Jaewook Shin's compiler which is described in section 2.5.2.4, should allow the programmer to write programs without worrying about using parallel constructs, and let them specify how the instructions are used by modifying the output from the EXPAND compiler, but the tool has to be both reliable and effective for this approach to be useful.

We have chosen, on account of the analyses, to make a data parallelism programming language intended for large computations that is inspired by NESL and ZPL, while maintaining a C-like syntax as seen in Mono's implementation. This should allow the programmer to focus on problem solving, instead of the actual implementation, while still utilising the possibilities of the target platform. As such, our language is intended as an abstraction from the low-level implementation of the instructions that facilitate data parallelism.

3.2 Compiler implementation language

We have chosen Java as the implementation language for our compiler simply due to the *Language and Compilers* course we are following alongside this project. The course encourages us to use Java, because Java is the language predominantly used when illustrating and introducing various concepts throughout the course. Furthermore the compiler generation tools we have been introduced to in the course have mainly been in their Java versions. These tools, and a couple more, are described in section 2.6.

None of us have any previous experience with Java and this project therefore provides a golden opportunity to familiarise ourselves with Java and take some of the theory regarding the compiler tools from the course and employ it in practice.

3.3 Target language

We have chosen to use the C programming language and GCC as the target platform for our compiler. This is because we already are familiar with the language and the use of GCC allows us to write assembly code interchangeably with ordinary C code, using its extended inline assembler extension, as seen in section 2.2.1. Using this as our target platform has several advantages:

- 1) It gives us easy access to SSE2 instructions.
- 2) The compiler handles most stack management.
- 3) Non-parallelisable code can be implemented using C.

Furthermore C is often regarded as “the universal assembly language” [31] due to the efficiency and low-level nature of the language. However the main argument for choosing C as target language is the support for embedded assembly code, which allows us to combine the low level instructions with some higher level C constructs.

We have chosen to only focus on the 64-bit version of GCC , thus we will only provide support for the 64-bit platform. Even though the compiler will be written in Java, we do not want to provide support for any operating systems beyond 64-bit Linux. Thus we do not intend to support software emulation or ports of GCC on Microsoft Windows, Mac OS X, Haiku, etc.

3.4 Parser generator

Our main reasons for using Coco/R are that it contains all the functionality we need in one program, is able to create Java code, has an easy-to-use input format, and creates more readable code than most of the alternatives we have looked at in section 2.6.

The reason for wanting a single tool to handle every aspect of the lexer and parser generation process is that multiple programs with different input and output formats seem like an unnecessary hassle, as changes in the language might require changes in multiple different files, and then running them through their respective tools. Having the entire language in one file also makes it more manageable to correct errors, as the entire definition can

be seen. Coco/R uses ENBF as input, which makes it both very easily readable and writable, as we have previous experience using this format from our courses. The tool has to support Java, as it is our implementation language of choice, as described in section 3.2. We want the tool to create code that we are able to relatively easily understand in order to make it easier to integrate it with the rest of the compiler, which would have been much harder to do if the generated code was hard to understand.

The main drawback of using Coco/R is that it creates a recursive descent parser from an LL(1) grammar. This puts some constraints on our grammar, compared to other classes of languages, since LL(1) is a fairly small class of languages. However, the language constructs for our language are possible to handle with an LL(1) parser with only a few workarounds, so this is not a hindrance. Coco/R also does not automatically create an AST; this has to be done manually by writing Java code inside the grammar. This obfuscates the grammar, but saves us from writing it again every time the parser has to be regenerated, due to errors or changes in the language.

Language design

In this chapter we are going to document the language we have designed and which we call CIP Computing in parallel (CIP). First we will introduce our language philosophy to make clear what we want to achieve and what the general idea behind the language is. Then parts of the operational semantics are reviewed, which consists of the abstract syntax for our language and the array semantics that we have chosen to implement. After this we will examine our language's formal and informal grammar, our choice of scope and type rules, then go through interesting constructs with some code examples. Lastly, this chapter will cover some of our previous design ideas that have been modified or removed and why this modification or removal was performed.

4.1 Design philosophy

The main focus of our language is to utilise data parallelism without encumbering the programmer with special keywords to specify parallel behavior, and let them use a syntax that is similar to many of the existing programming languages as discussed in section 3.1. This allows for the programmer to focus on being productive and minimize the time needed to learn the new language.

This is done mainly by using a C-like syntax with procedures, assignments and declarations, but made more verbose to enhance readability by using keywords to specify the beginning and end of the various construct, a feature inspired by the ZPL programming language. Another design choice directly influenced by the use of data parallelism is how our language handles arrays as they are considered first-class citizens as inspired by ZPL and therefore can be used as parameters for procedures and assigned to variables. One exception is that arrays can not be sent as return values from procedures. This is due to implementation issues that are further discussed in section 7.2. These variables can then be used directly in computations by specifying the indexes involved. How these computations then are executed on the underlying platform is handled by the compiler.

4.2 Operational semantics for CIP

In this section we will describe the semantics for arrays in CIP. We will also describe the abstract syntax and the environment-store model, both of which are used in the description of the semantics.

4.2.1 The environment-store model

In our semantics we will use the environment-store model to describe the program state. This model is a product of the work by Dana Scott and Christopher Strachey on denotational semantics [32]. The environment-store model can be regarded as an abstract model of a computer in that it describes how variables and procedures are actually bound during a program execution: Every variable is bound to a storage cell and the value of a variable is the content of a storage cell [32].

The environment-store model makes use of three different sets to describe the binding of variables:

1. **Var** \cup *next* is the set of variables and *next* is a special pointer to the next available location.
2. **Loc** is the set of locations.
3. **Sto** (shorthand for store) is the set of values.

We will distinguish between *variable environment* and *procedural environment* however they may both be regarded as functions given an argument that returns the storage location to which it is bound [32]. One may think of this function as a symbol table. The *store* is a function that takes a location as input argument and returns the corresponding value found at that location. Figure F4-1 gives a visual interpretation of the environment-store model. The figure shows that a variable maps to a location which in turn

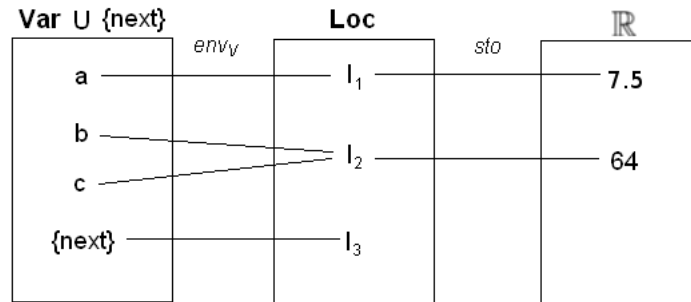


Figure F4-1: Visual interpretation of the environment-store model

maps to a value. Furthermore it shows that aliasing is possible, as two variables may map to the same location. In our language that would be possible if the variables are declared as arrays.

4.2.1.1 Sets and functions definitions

In this section we will describe and define the sets and functions that we use to describe the semantics of our programming language.

Locations In the environment-store model we call storage cells *locations*. We let **Loc** be the set of locations. Further we let l denote an arbitrary element of **Loc**. For simplicity we will always assume that locations are natural numbers, such that **Loc** = \mathbb{N} . [32]

The special pointer *next* points to the next available location. This pointer helps make sure that we do not unintentionally overwrite the content of a storage cell when allocating new variables.

With regard to **Loc** we will define the function *new* which is a total binary function that given a location returns its successor

$$new : \mathbf{Loc} \rightarrow \mathbf{Loc}$$

Due to our definition of **Loc** as consisting of natural numbers, *new* may be defined by

$$new(l) = l + 1$$

Environments A *variable environment* contains the symbolic names of variables. We define the set of variable environments as a set of partial functions from variables to locations:

$$\mathbf{EnvV} = \mathbf{Var} \cup next \rightarrow \mathbf{Loc}$$

Furthermore we let env_V denote an arbitrary member of **EnvV**. Given a variable name as input env_V returns its location. [32]

A *procedural environment* contains the symbolic names of procedures. We define the set of procedural environments as a set of partial functions that map procedure names to the Cartesian set of statements, variable environments, and procedural environments:

$$\mathbf{EnvP} = \mathbf{Pnames} \rightarrow \mathbf{Stm} \times \mathbf{EnvV} \times \mathbf{EnvP}$$

This definition implies static scope rules since **EnvV** and **EnvP** are the environments known at declaration time.

We let env_P denote an arbitrary member of **EnvP**. Given a procedure-name as input it returns a triple.

Store A *store* simply contains values. We define the set of stores as a set of partial functions that map locations to values:

$$\mathbf{Sto} = \mathbf{Loc} \rightarrow \mathbb{R}$$

Since we only have integer and floating-point data types in our language we let stores only map locations to real numbers. In reality the set of real numbers is far greater than the set of values representable by our language's data types. Mainly because \mathbb{R} is infinite, whereas the numbers in our language are limited by hardware considerations. However for the sake of simplicity we will use \mathbb{R} as the range of values.

4.2.2 Abstract syntax

In this section we provide an abstract syntax for CIP. The abstract syntax will be an abstraction of our CFG from appendix A. We will use a notation similar to the one in [32]. Table T4-1 shows the syntactic categories for CIP and table T4-2 shows the abstract syntax.

<i>Syntactic categories</i>	
$n \in \mathbf{Num}$	Numerals
$x \in \mathbf{Var}$	Variables
$s \in \mathbf{String}$	Strings
$y \in \mathbf{Array}$	Arrays
$a \in \mathbf{Arith}$	Arithmetic expressions
$A \in \mathbf{ArrayArith}$	Array arithmetic expressions
$S \in \mathbf{Stm}$	Statements
$p \in \mathbf{Pnames}$	Procedure names
$E \in \mathbf{Expression}$	Expressions
$D_P \in \mathbf{DeclP}$	Procedure declarations
$D_A \in \mathbf{DeclA}$	Array declarations
$D_V \in \mathbf{DeclV}$	Variable declarations

Table T4-1: Syntactic categories for CIP.

<i>Formation rules</i>
$a ::= n \mid x \mid (a_1) \mid y[a]$ $\mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2 \mid a_1 / a_2 \mid a_1 \bmod a_2$ $\mid a_1 \mathbf{AND} a_2 \mid a_1 \mathbf{OR} a_2 \mid \neg a_1 \mid a_1 = a_2 \mid a_1 \neq a_2$ $\mid a_1 > a_2 \mid a_1 \geq a_2 \mid a_1 < a_2 \mid a_1 \leq a_2$
$A ::= \mid A_1 + A_2 \mid A_1 - A_2 \mid A_1 \cdot A_2 \mid A_1 / A_2 \mid A_1 \bmod A_2$ $\mid A_1 \mathbf{AND} A_2 \mid A_1 \mathbf{OR} A_2 \mid \neg A_1 \mid A_1 = A_2 \mid A_1 \neq A_2$ $\mid A_1 > A_2 \mid A_1 \geq A_2 \mid A_1 < A_2 \mid A_1 \leq A_2 \mid (A_1) \mid y \mid y[a_1] :: [a_2]$
$E ::= a \mid A$ $\mid A_1 + a_2 \mid A_1 - a_2 \mid A_1 \cdot a_2 \mid A_1 / a_2 \mid A_1 \bmod a_2$ $\mid A_1 \mathbf{AND} a_2 \mid A_1 \mathbf{OR} a_2 \mid \neg E_1 \mid A_1 = a_2 \mid A_1 \neq a_2$ $\mid A_1 > a_2 \mid A_1 \geq a_2 \mid A_1 < a_2 \mid A_1 \leq a_2 \mid (E_1)$
$S ::= x := a; \mid y[a] := a; \mid \mathbf{while} \ a \ S_1 \ \mathbf{end} \mid S_1 S_2 \mid y := a; \mid y := A;$ $\mid \mathbf{if} \ (a) \ S_1 \ \mathbf{else} \ S_2 \ \mathbf{end} \mid D_V \mid D_A \mid D_P \mid p(E) \mid E$ $\mid \mathbf{print} \ E \ ; \mid \mathbf{print} \ s \ ; \mid \mathbf{println} \ E \ ; \mid \mathbf{println} \ s \ ;$
$D_V ::= \mathbf{var} \ x := a; \mid D_V \mid \varepsilon$
$D_A ::= \mathbf{var} \ x[a] := \{a_1..a_n\}; \mid D_A \mid \varepsilon$
$D_P ::= \mathbf{procedure} \ p(\mathbf{var} \ x) \ S \ \mathbf{end} \mid D_P \mid \mathbf{procedure} \ p(\mathbf{var} \ y) \ S \ \mathbf{end} \mid D_P \mid \varepsilon$

Table T4-2: Abstract syntax for CIP.

We have chosen to use standard mathematical symbols within our abstract syntax, because it is supposed to be more compact and easier to read than our CFG. The operators ($!$, $==$, $<=$, $>=$, $\%$) have been replaced with (\neg , $=$, \leq , \geq , \bmod). Furthermore, we have replaced the assignment operator from our CFG with $(:=)$ to avoid confusion with the equality operator $(=)$ in our abstract syntax.

Concrete data types have been replaced with the keyword **var** which can mean any of the int and float data types in CIP.

4.2.3 Operational semantics for arrays in CIP

In this section we will describe the semantics for arrays in CIP. We will also show the semantics for procedure calls, since procedure calls with scalars as parameters have different semantics from procedure calls with arrays as parameters. The semantics described in this section are based on the abstract syntax given in section 4.2.2. In the semantics given here, we abstract from some details and some operations are simplified, and these will be discussed where applicable. For a full list of the semantic rules that we have chosen to describe formally, see appendix C.

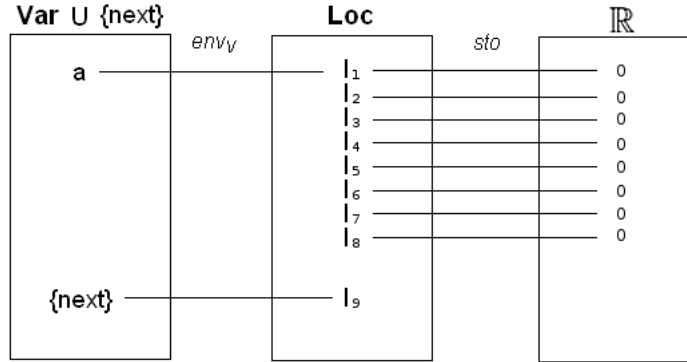


Figure F4-2: Visual interpretation of the environment-store model with arrays

When array names are looked up in the variable environment, it returns a single location, and the rest of the elements in the array must be accessed through adding an index to the base location of the array. This can be seen in the rules in figure F4-3 and figure F4-4.

In general, the semantics for arrays are that operations are performed in parallel on a number of data elements at a time. The exact number of data elements that are worked on at a time depends on the data type being worked on. In the formal semantics given here, we will assume that four data elements are being worked on at a time. It is straightforward to extend this to more or fewer elements. Since both ints and floats are described with the **var** keyword, we do not differentiate between the two in our semantics. In our actual implementation, however, only ints are allowed as array indexes.

Data parallelism in CIP is only utilised when working on a row in an array. This can be a part of a row, but it can not be multiple rows. If a single element from an array is being worked on, the semantics are equivalent to the semantics for scalars. In the semantics given here, we will only describe arrays of one dimension, but it is straightforward to expand this description to include multiple dimensions as long as one keeps in mind that the parallel operations on arrays only work on rows. CIP includes a special range operator $::$ which means that the array it is used on is only being worked on from the element at the index of the left-hand side to the element

$$[ArrayLoc] \quad env_V, sto \vdash y \rightarrow_A l \quad \text{where } env_V y = l$$

Figure F4-3: Semantic rule for retrieving the location of an array.

$$[ArrayIndex] \quad \frac{env_V, sto \vdash y \rightarrow_A l}{env_V, sto \vdash y[a] \rightarrow_a v} \quad \text{where } sto \ l + a = v$$

Figure F4-4: Semantic rule for indexing an element in an array.

$$[RangeOp] \quad \frac{env_v, sto \vdash y \rightarrow_A l}{env_v, sto \vdash y[a_1] :: [a_2] \rightarrow_A l'} \quad \text{where } l' = l + a_1$$

Figure F4-5: Semantic rule for using the range operator.

at the index of the right-hand side. As an example,

$$A = B[2] :: [5]$$

assigns only the elements from index one to five of array B to array A. The semantics of this operator can be seen in figure F4-5. Apart from looking up the first location of the selected subarray, the compiler also sends information about the length of the subarray, so that the code generator can work on the correct number of elements.

The data parallelism of CIP can be seen in several of the semantic rules. Looking at figure F4-6 and figure F4-7, we can see that when an array is assigned to another array, four elements of the array being assigned from are assigned to the array being assigned to at a time. Likewise, when a scalar is assigned to an array, the scalar is assigned to all of the elements of the array, four elements at a time.

For arithmetic expressions, we can see from figure F4-8 that when a scalar is added to an array, this operation applies the scalar to all the elements of the array, four elements at a time. Similarly, we can see from figure F4-9 that arrays are added to one another element by element in blocks of four at a time. It is straightforward to extend these rules to include rules for the remaining arithmetic expressions as they are similar except for the operation being performed.

$$\begin{aligned}
& env_V, env_P \vdash \langle y := A, sto \rangle \rightarrow_s sto[l_1 \mapsto v_1][l'_1 \mapsto v_2][l''_1 \mapsto v_3][l'''_1 \mapsto v_4] \\
& \text{where } l_1 = env_V \ y \text{ and } sto \ l_2 = v_1 \\
[ArrayArrayAss] \quad & \text{and } l'_1 = new \ l_1 \text{ and } l'_2 = new \ l_2 \text{ and } sto \ l'_2 = v_2 \\
& \text{and } l''_1 = new \ l'_1 \text{ and } l''_2 = new \ l'_2 \text{ and } sto \ l''_2 = v_3 \\
& \text{and } l'''_1 = new \ l''_1 \text{ and } l'''_2 = new \ l''_2 \text{ and } sto \ l'''_2 = v_4
\end{aligned}$$

Figure F4-6: Semantic rule for assigning arrays to other arrays.

$$\begin{array}{l}
env_V, env_P \vdash \langle y := a, sto \rangle \rightarrow_s sto[l \mapsto v][l' \mapsto v][l'' \mapsto v][l''' \mapsto v] \\
[ArrayScalarAss] \quad \text{where } env_V, sto \vdash a \rightarrow_a v \text{ and } env_V y = l \\
\text{and } l' = new\ l \text{ and } l'' = new\ l' \text{ and } l''' = new\ l''
\end{array}$$

Figure F4-7: Semantic rule for assigning scalars to arrays.

$$\begin{array}{l}
\frac{env_V, sto \vdash A \rightarrow_A l \quad env_V, sto \vdash a \rightarrow_a v_2}{env_V, sto \vdash A + a \rightarrow_E V} \\
\text{where } v_1 = sto\ l \text{ and } v[0] = v_1 + v_2 \\
[ArrayScalarAdd] \quad \text{and } l' = new\ l \text{ and } v'_1 = sto\ l' \text{ and } v[1] = v'_1 + v_2 \\
\text{and } l'' = new\ l' \text{ and } v''_1 = sto\ l'' \text{ and } v[2] = v''_1 + v_2 \\
\text{and } l''' = new\ l'' \text{ and } v'''_1 = sto\ l''' \text{ and } v[3] = v'''_1 + v_2
\end{array}$$

Figure F4-8: Semantic rule for adding scalars to arrays.

$$\begin{array}{l}
\frac{env_V, sto \vdash A_1 \rightarrow_A l_1 \quad env_V, sto \vdash A_2 \rightarrow_A l_2}{env_V, sto \vdash A_1 + A_2 \rightarrow V} \\
\text{where } v_1 = sto\ l_1 \text{ and } v_2 = sto\ l_2 \text{ and } V[0] = v_1 + v_2 \\
\text{and } l'_1 = new\ l_1 \text{ and } l'_2 = new\ l_2 \\
[ArrayArrayAdd] \quad \text{and } v'_1 = sto\ l'_1 \text{ and } v'_2 = sto\ l'_2 \text{ and } V[1] = v'_1 + v'_2 \\
\text{and } l''_1 = new\ l'_1 \text{ and } l''_2 = new\ l'_2 \\
\text{and } v''_1 = sto\ l''_1 \text{ and } v''_2 = sto\ l''_2 \text{ and } V[2] = v''_1 + v''_2 \\
\text{and } l'''_1 = new\ l''_1 \text{ and } l'''_2 = new\ l''_2 \\
\text{and } v'''_1 = sto\ l'''_1 \text{ and } v'''_2 = sto\ l'''_2 \text{ and } V[3] = v'''_1 + v'''_2
\end{array}$$

Figure F4-9: Semantic rule for adding an array to another array.

4.3 Our language

In this section we are going to look at the interesting constructs that CIP provides the programmer with. The language we have made is much like C and ZPL, as it is loosely based on the two and on the considerations in section 3.1. The CFG of CIP can be found in appendix A. For full examples of programs written in CIP, see appendix B. The following sections use fragments from the code in the appendix to exemplify the discussion.

4.3.1 Arrays in CIP

All operations on more than a single element in CIP of an array must be done using the range operator `::`. This construct provides the programmer with the ability to operate on selections of multidimensional arrays as is shown in source code 4.1. This operation is done using the operator `::` with the from-selection operand on the left side and the end-selection operand on the right side.

```

1  int8 Array2[3] = {4,5,6};
2  int8 Vector[3][1] = {{7},{7},{7}};
3  int8 Matrix[3][3] = {{1,1,1},{1,1,1},{1,1,1}};
4
❶ 5  Matrix[2][0]::[2] = Array2[0]::[2];
❷ 6  Matrix[1][0]::[2] = Matrix[0][0]::[2] *
7      Matrix[2][0]::[2][2];

```

Source code 4.1: Multidimensional array indexing in CIP.

The indexing operand describes a specific location in an array by denoting a specific element in each dimension as seen in source code 4.1. When used together with the `::` operator, this denotes an area of the array that is stretched between the two locations. This area must be a subsection of a single array dimension, so the range operator can not be used across several dimensions.

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

Figure F4-10: Multidimensional array selection example.

The grey area in figure F4-10 is a selection made to the multidimensional by the command `Matrix[0][0]::[2]`. By using this range operator, it is possible to use binary operators with arrays of different dimensions as operands, because it is possible to select an appropriate part of an array, thus allowing the programmer to operate on a smaller part of a larger array. In source code 4.1, we operate on a part of the array *Matrix* ❶, where the array *Array2* is assigned to the bottom part of *Matrix*, as shown in figure F4-11.

After this we multiply the bottom row in *Matrix* with the top row elements and assign the result to the middle row of the array ❷, as shown in figure F4-12.

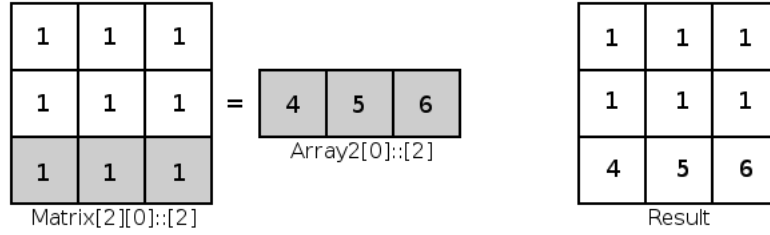


Figure F4-11: Assigning a small array to the bottom row of another array.

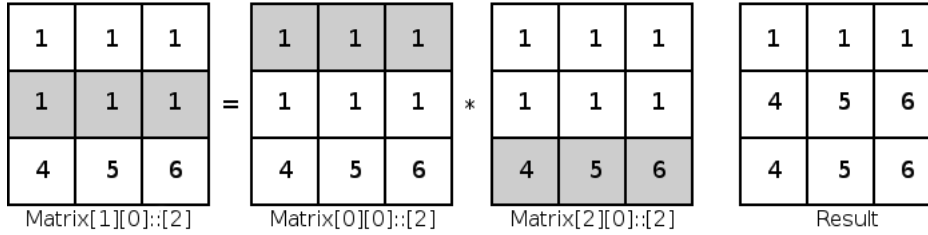


Figure F4-12: Multiplying parts of an array with itself in CIP.

4.3.2 CIP binary operators

The second interesting construct we are going to document is how it is possible to use binary operators with arrays as seen in source code 4.2 with the arithmetic multiply operator and the relational less-than operator as examples. The binary operators in CIP operate on each individual element of the arrays and return an array of the same size with each element corresponding to the result of the operation. To use binary operators with arrays as operands, the array dimensions must be equal.

```

1  int8 Array1[3] = {1,2,3}, Array2[3] = {4,5,6};
2
3  //assigns {4,10,18} to Array1.
4  Array1 = Array1 * Array2;
5
6  //assigns {0,1,1} to Array1.
7  Array1 = Array1 > Array2;
```

Source code 4.2: Binary operators with arrays as operands in CIP.

4.3.3 CIP unary operators

The last construct we are going to describe is how it is possible to use unary operators with arrays as seen in source code 4.3 with the arithmetic negation operator and the logic not operator as examples. The unary operators in CIP operate on each individual element of the arrays and return an array of the same size with each element corresponding to the result of the operation.

```

1  int8 Array1[3] = {0,1,1};
2
3  //returns array[3] = {1,0,0}.
4  Array1 = !Array1;
5
6  //returns array[3] = {-1,0,0}.
```

```
7 Array1 = -Array1;
```

Source code 4.3: Binary operators with arrays as operands in CIP.

4.4 Contextual rules

In this section we will define the type and scope rules for our language.

4.4.1 Type rules

In our programming language we have two primitive types, `int` and `float`, and one composite type, `array`.

Both the primitive types can have different sizes. `Int` types can be `int8`, `int16`, `int32`, or `int64`, where the number denotes the size of the data type in bits. If declared with no suffix, an `int` will have a size of 32 bits. More information about our ints can be found in table T4-3. `Float` types can be `float32` or `float64`, with the default being 32 bits. More information about the float datatypes is found in table T4-4. Arrays can contain all the different primitive datatypes, but only one defined datatype at a time.

We have chosen these primitives as datatypes for our language because they have to match the SIMD instructions which operates on datatypes of these sizes [8]. It is logical that it would be an unnecessary workload to convert the datatypes from incompatible types to SIMD compatible types.

Bits	Name	Range	Decimal Digits
8	<code>int8</code>	$-(2^7)$ to $2^7 - 1$	3
16	<code>int16</code>	$-(2^{15})$ to $2^{15} - 1$	5
32	<code>int32</code>	$-(2^{31})$ to $2^{31} - 1$	10
64	<code>int64</code>	$-(2^{63})$ to $2^{63} - 1$	19

Table T4-3: Overview of the different int sizes in CIP on a 64-bit Linux machine using GCC.

Bits	Name	Exponent	Significand	Exponent bias	Bits precision
32	<code>float32</code>	8	23	127	24
64	<code>float64</code>	11	52	1023	53

Table T4-4: Overview of the different float sizes in CIP on a 64-bit Linux machine using GCC.

4.4.1.1 Type checking

Type checking is ensuring that the operands of an operator are compatible types. A compatible type is one that is legal for the operator or allowed under the language to be implicitly converted by the compiler in the generated code to a legal type. Such an implicit approach is called coercion. An example could be if an `int` and a `float` variable are added in Java, the value of the `int` is coerced to a `float` and a floating-point addition is done. If an operand is not a compatible type, a type error occurs.

There are two different kinds of type checking: static type checking and dynamic type checking. Static type checking reduces the risk of run-time errors since the types are statically bound and cannot change throughout the program. They will be checked at compile-time and if any errors are detected, the program will not compile.

On the other hand, dynamic type checking allows for more flexibility for the programmer, because the types can be changed at run-time as they are dynamically bound, but can lead to more run-time errors if not used carefully.

Since we have no run-time environment for our programming language, we use static type checking and static type binding. In order to avoid both loss of precision and data overflow, we do not allow either explicit or implicit casting between our two primitive types. Within each primitive type, however, we allow implicit casting from a smaller bit representation to a larger bit representation, for example from an `int8` to an `int16`. We do not allow casting from a larger bit representation to a smaller.

4.4.2 Scope rules

Like with type checking and binding, there are two different kinds of scope rules: static scope rules and dynamic scope rules. With static scope rules, a procedure is called with the bindings that were known when the procedure was declared, whereas dynamic scope rules use the bindings known when the procedure is called. [32] This is shown as pseudo-code in source code 4.4.

```
1   String y = "global";
2
3   function print_y() {
4       print(y);
5   }
6
7   function test_scope() {
8       String y = "local";
9       print_y();
10  }
11
12  test_scope();
13  //Statically scoped languages print "global".
14  //Dynamically languages print "local".
15
16  print_y();
17  //All languages should print "global".
```

Source code 4.4: Static and dynamic scope rule example in pseudo-code [33, 31].

In our language we use static scope rules. In addition, it is possible in our language to look outside a scope to higher scopes, but not to look into lower scopes. So if a variable `y` is used in a scope where it is not declared, the compiler will look for a variable `y` in the outer scopes, and use that `y` which is in the lowest of its outer scopes. This also means that side effects are possible in our functions.

4.5 Language changes

Changes to the language were made throughout the project, either because the original construction had some undesirable aspects, or because they became unnecessary in their current form as the language evolved.

The following section describes the various changes made to the language, as part of the development process, with the reasoning behind the changes and the effect they had on the overall language as its primary focus.

4.5.1 Redesigned features

The following elements in the language were changed after their initial design was finished. This was due to the introduction of procedures in the language, which effected some already established elements, along with problems uncovered while developing programs in the language.

4.5.1.1 Array declaration

Declaration of arrays was originally done using the `dims` keyword. This allowed assignments to specific elements in the array to be done at the declaration. The following would, as an example, assign four to the third element of a one dimensional array of ten integers:

```
int dims[10] SmallArray[3] = 4;.
```

This was contrary to many of today's popular languages like C and Java [34], where arrays declarations does not require a special keyword, just that their dimensions are defined as suffix to their name. This of course does not allow the above assignment, but as most uses of arrays require that all of its elements are declared, and not just a single element, we decided that it would be preferable to make array declarations more akin to the popular languages in order that existing programmers may learn it more quickly.

4.5.1.2 Array dimension

Specifying the dimensions of an array was originally done using square brackets, where the size of each dimension was separated with a comma, so for example a $3 \times 3 \times 3$ array would be written as `[3,3,3]`. This unfortunately was a problematic construction when used to only define dimensions, as was needed when an array was used as parameter for a procedure. As the following example shows, it was unnecessarily difficult to quickly see the number of dimensions of an array, when the numbers denoting the precise size was not present:

```
procedure int ArrayElementSum(int[, ,] input).
```

The procedure would accept any array with three dimensions, and return a single int. The use of square brackets and commas to denote this, although consistent with the declaration of dimensions in arrays, gives the immediate impression that the procedure accepts a two-dimensional array since there are two commas. The problem is further worsened when arrays with many dimensions are used, as it can be hard to determine the number of commas effortlessly. To alleviate this problem, we decided to use sets of square

brackets to denote the number of dimensions. Using this method the above procedure would be

```
procedure int ArrayElementSum(int[] [] [] input).
```

This makes it much easier to read, because of its larger size, at the cost of some writeability. This change also makes our language more like popular languages already in use, which should make learning our language easier for existing programmers [34].

4.5.1.3 While loop

As our language originally contained multiple loop constructs as described in section 4.5.2.1, we decided to unify them by prefixing them all with the **repeat** keyword. But after the **repeat until** loop was removed from the language, the use of the **repeat** keyword became unnecessary as the language now only contained a single loop construct. We then decided to change the while loop from **repeat while** to just **while**, which made it quicker to write and still keeps it obvious what the functionality of the construction is.

4.5.1.4 Multidimensional ranges

Originally we wanted to allow array operations on multiple dimensions of an array at a time instead of just on a single row at a time as is currently required. If multidimensional ranges were allowed, it would be possible for example to assign a 2×2 matrix to another 2×2 matrix using the following syntax:

$$A[0][0] :: [2][2] = B[0][0] :: [2][2]$$

With this syntax it would be possible to specify an arbitrary subarray with an arbitrary number of dimensions and would allow great flexibility in working with arrays. This feature was changed to only allow ranges to be specified for a single dimension because extra overhead was introduced when extracting the subarrays in a manner that made it possible to work on them with SIMD instructions and this extra overhead went contrary to the extra execution speed that we wanted to obtain by using SIMD instructions. Issues with the way we chose to implement the compiler also made it cumbersome to acquire the information required to extract the subarrays. These issues are further discussed in section 7.2.

4.5.2 Other ideas

Three features initially thought of as candidates in the language, but were later removed from the drawing board. These were the **repeat until** loop, the assignment of a range of numbers to a collection and using the range operator on several dimensions.

4.5.2.1 Repeat loop

The loop followed the implementation used by Pascal, which has the form **Repeat** *S* **Until** *E*. This construction repeats the statement *S* until the boolean expression *E* equals true [35]. This construction was removed mainly to simplify the language, as it contained both a repeat until, and a repeat while loop. And as both loop constructions are equally powerful in their

expressive capabilities [36], since a negation of the boolean expression in a while loop would have the same result, we did not see any reason to have our language contain both. We decided to remove the **Repeat S Until E** instead of the while version, as most of the popular programming languages used today use a while loop construction [34], so using this kind of loop construction in our language would make it more familiar to most programmers.

4.5.2.2 Ranges

The purpose of ranges was to simplify the assignment of large sets of numbers to arrays. The implementation was as follows, where A is an array comprised of ten integers, which is assigned the numbers from one to ten:

```
int A[10] = [1..10].
```

There were several problems with this, however. One problem was that the semantics were not clear. What does

```
float A[5] = [2.46..5]
```

mean? One possible interpretation would be that the interval between 2.46 and 5 would be split into four equal parts, and the numbers between these parts would be assigned to the array. So the given example would produce an array

```
{2.46, 3.095, 3.73, 4.365, 5}.
```

This would also be a problem with ints when the interval is too small or too large for the array size. As an example, observe the array declaration

```
int[3] = [1..100].
```

Again we could interpret this as giving an array

```
{1, 50, 100},
```

but this interpretation becomes problematic when the computed numbers can not be represented as integers.

Another problem was that the implementation became convoluted when used with arrays of more than one dimension, as the order in which the different dimensions of the array was initialised had to be understood by the programmer in order to use it correctly. One possible solution to this, was to implement it similar to Haskell [37], and thereby limit the use of ranges to one dimensional arrays. But as the construction had lost most of the convenience we originally had intended by using this implementation, and because of the unclear semantics, we decided not to implement it in the first version of the language, as we believe a more general solution at a later point would be a better solution.

Compiler architecture

This chapter documents the overall design of our compiler and the choices we made during its construction. This is done through abstract diagrams of the entire compiler structure, and small examples of specific interesting elements. Our diagrams use the notation documented in [38]. Documentation of the various design patterns used in the compiler can be found in section 5.1. The use of `/* ... */` in source code listings indicates that code unnecessary for the explanation has been omitted for simplicity.

5.1 Design patterns

As multiple design patterns are used in the compiler, some of which have had an impact on the overall design, the following section will document how each of these patterns work and what problems they solve.

5.1.1 Singleton pattern

The purpose of the singleton pattern is to restrict instantiation of the class it is implemented in so that only one instance is available at any given time and no more can be created. This is done by restricting access to the constructor of the class and letting a getter handle instantiation of the class [39].

```
1 class Singleton {
2
3 /* Getter for the instance */
4 ① public static synchronized Singleton getInstance() {
5     if (instance == null)
6         instance = new Singleton();
7
8     return instance;
9 }
10
11 /* Private constructor */
12 ② private Singleton() {
13 }
14
15 /* Private static instance */
16 ③ private static Singleton instance = null;
17
18 }
```

Source code 5.1: Singleton example. [40, 41]

The pattern consists of three elements. The first is a static variable for holding the instance of the class ③. This together with the constructor is

declared as private to prevent instantiation from outside the class ❷. All instantiation is done through the `getInstance` method ❶. This ensures that the class is instantiated if the private variable at ❸ is null whereas if it has already have been assigned a value then this value is returned.

5.1.2 Factory pattern

The purpose of the factory pattern is to hide the instantiation logic of similar objects inside a class and use the newly created class through a common interface. This common interface removes the need for other elements of the program to know how the object was created. The use of a factory removes the need for classes to know how to create objects of different types. This pattern also ensures that the instantiation of the objects only exists in one place of code which allows for changes in how objects are instantiated without concern for other classes. [40, 41]

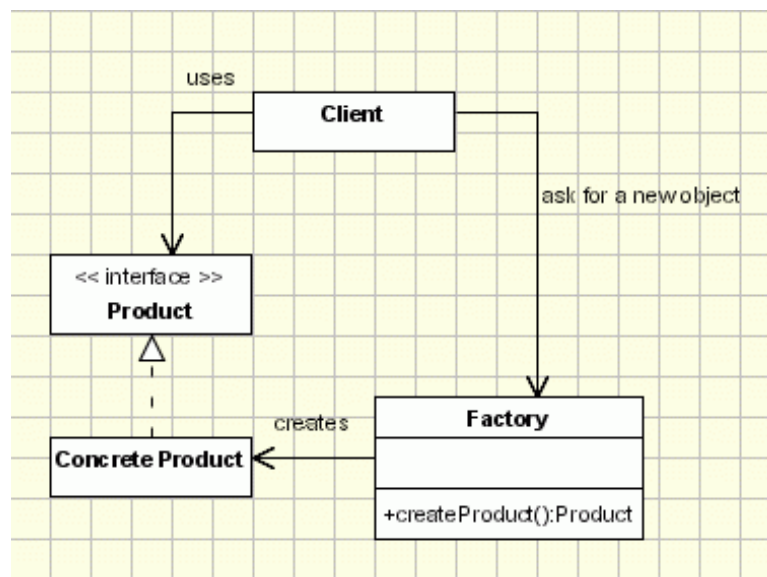


Figure F5-1: A simple implementation of a factory [40].

The pattern in the simple implementation shown in figure F5-1 only consists of a few elements. The **Factory** class contains the logic needed to instantiate classes which all implement the **Product** interface. This interface defines the methods needed to use the different products the factory is capable of producing. This allows the client to request an object from the factory with no concern for how it is created. It also lets the clients use it through its interface with no need to know about how the product implements the operations. Use of the pattern thereby heightens encapsulation by minimizing the need for the client to know anything about the product apart from the operations that the interface implementation defines. [40]

5.1.3 Visitor pattern

The visitor pattern allows for operations working on an object structure instead of implementing it together with the structure itself. This allows for similar types of operations to be bundled together which allows for better code reuse and encapsulation as helper methods can be implemented together with the methods that use them. Implementing the code directly on

classes could lead to redundant code if similar operations are needed multiple places in the collection. Restructuring the collection to give these elements a common ancestor is not possible [41, 39, 40].

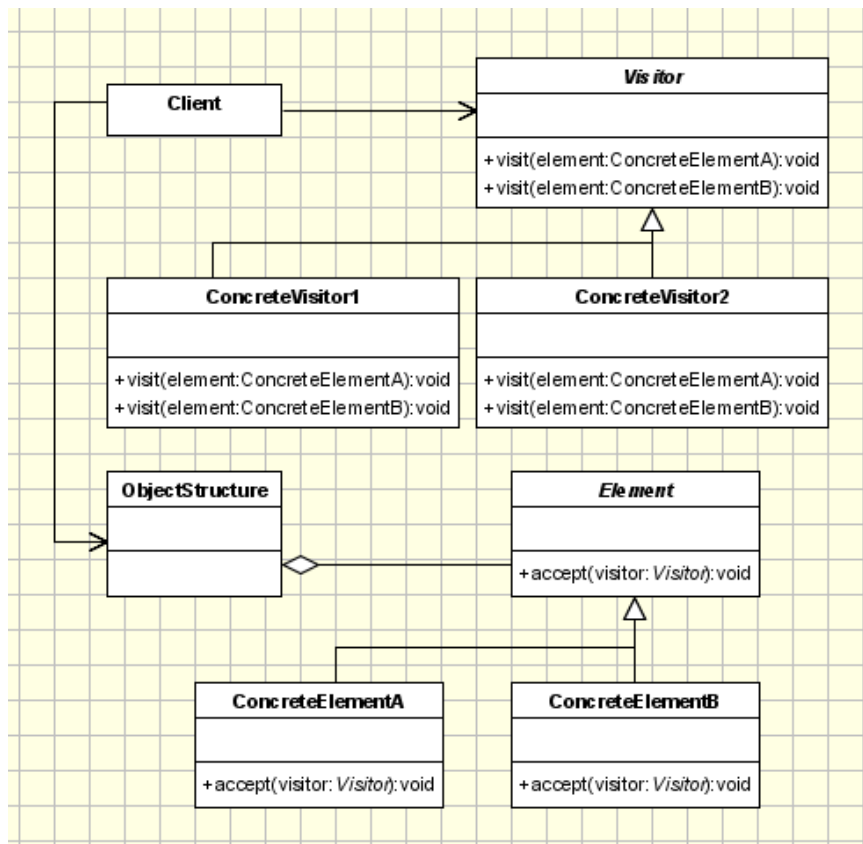


Figure F5-2: A simple implementation of the visitor pattern [40].

Multiple elements work in conjunction in the visitor pattern. First the object structure that the visitor is going to visit needs to implement the **accept** method. This method functions as the interface for the visitors on the object structure and allows an element in the object structure to accept a visitor so it can be visited. The accept method should be implemented at the top of the inheritance hierarchy. Optionally it can be defined in an interface in order to make its presence more explicit.

The visitor itself is divided into an abstract visitor and a number of concrete visitors. The abstract visitor defines the elements in the collection that the concrete visitors need to implement **visit** methods for, while the actual operations on the object structure is defined in the concrete visitors. This way performing a new operation on the object structure only requires a new visitor to be constructed as it is not necessary to make any changes to the structure itself. [41, 39, 40].

5.1.3.1 Reflection

For all its benefits the visitor pattern also has a major drawback, namely that all concrete visitors need to implement every **visit** method defined in the abstract visitor. This means that even if some visitors only need to visit some elements and is not concerned with the other elements that the object structure contains, it must still implement visit methods for all el-

ements. Implementing a new type of object in the structure forces every visitor to implement a **visit** method for this new type before testing can be done [40]. This problem can fortunately be alleviated by the use of reflection in languages which support it like Java and C#, at the cost of reduced performance.

Reflection allows the concrete visitors to iterate through their available methods and use the most appropriate or call a default **visit** method if a specialised version is not available. This allows for the implementation of new objects in the structure being visited without the need for all visitors to implement a visit method for this type. The visitors will use the one that is the most appropriate based on the methods that are available, based on which classes the element being visited inherits from and what interfaces it implements [40].

5.2 Compiler model

The compiler is separated into a set of different components through simple interfaces, as illustrated in figure F5-3. This allows us to hide the implementation of the class, which can be changed as long as the interface of the classes stays the same. This creates a decoupling between the different components, which makes it simpler to work simultaneously on the different components, as other modules only depend on the exposed interface. This also means that the dependencies between the components in figure F5-3 should be understood as dependent on the interface of the component, and not its actual implementation.

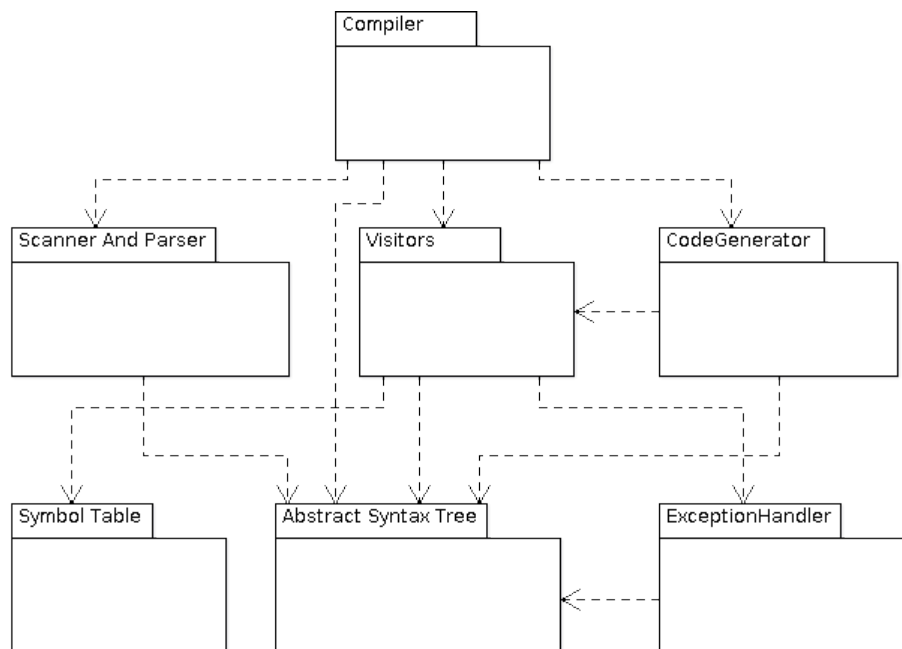


Figure F5-3: An abstract model of our compiler.

Another very important function of the module-based approach we have taken to the overall design of the compiler is that it encapsulates responsibility. This makes it very clear where new functionality should be implemented,

and what part of the program that contains the code to debug when bugs are found. This in turn gives more freedom to the programmer doing development, as only the interface of the modules have to be defined collectively, since the implementation of the component is the programmer's responsibility, and the responsibility of the module is defined through the model.

5.2.1 Compiler

This component consists of two classes: One named **Compiler** and another named **Utils**. The **Utils** class contains various methods needed throughout the compiler, for example to convert data types to strings or find the highest order bit that is set to one. The **Compiler** class contains the main subroutines and executes the various parts of the compilation process, based on the command line parameters it has received. This is mainly done through two methods: **parseCompileFlags** which parses the command line flags and configures the compiler accordingly, and **compile** which handles the compilation process.

The parsing of the command line flags in **parseCompileFlags** shown in source code 5.2 is a simple string comparison of the input with known flags formatted as a switch case.

```

1 private void parseCompileFlags(List<String> flagsList) {
2     for(String flag : flagsList) {
3
4         switch(flag) {
5             case "--Print-AST":
6                 this.flags[0] = true;
7                 break;
8                 /* ... */
9             default:
10                this.compileStatus = false;
11                printHelp();
12        }
13    }
14}

```

Source code 5.2: The parseCompileFlags method.

If the given string matches a known flag, then the corresponding boolean value in the control array is set to true indicating that this option has been enabled ❶. If the string does not match any cases in the switch, then the given flag is unknown to the compiler and the boolean *compileStatus* is set to false ❷, indicating that no more of the compilation should be executed. This in practice terminates the compiler, but allows for the compiler to terminate gracefully as the compilers help page is printed to the user.

The compilation process itself is controlled by the **compile** method, which can be seen in source code 5.3. This method ensures that the process is done according to the command line flags, that errors are handled properly, and it makes controlling the process more manageable overall.

```

1 public boolean compile() {
2     if(compileStatus)
3         scanAndParse();
4
5     if(compileStatus)
6         declarationAnalysis();

```

```

7
8      /* ... */
9
10     if(compileStatus && !flags[1])
11         codeGeneration();
12
13     printAllExceptions();
14
15     return compileStatus;
16 }

```

Source code 5.3: The compile method.

The boolean *compileStatus* is checked before executing each part of the compilation process ❶ in order to ensure that no errors have been found in any of the prior methods. This is done because errors in the earlier stages of the compilation in the CIP program might cause other errors later, which will make it harder for the programmer to see what the problem exactly is. Parts of the compiler can be omitted based on the given compile flags. For example, the code generation is omitted if the compiler is given the “no-code” flag ❷. At ❸ all errors and warnings produced by the various parts of the compiler are formatted and printed by **printAllExceptions**.

5.2.2 Scanner and parser

This component is, contrary to the other compiler components, not written by hand but instead generated using the Coco/R compiler generator. Our reasons for choosing this particular tool is documented in section 3.4. That is why we in this section will focus on how we generate the source code and only document the parts of the scanner and parser that are needed to illustrate this. The generated scanner and parser is based on a CFG which can be seen in appendix A.

```

1 /* ... */
2
3 ❶ Loop <out Node loop>      (. loop = null; Node cond, s,
   loopBlock; .)
4
5 =
6
7 ❷ "while"                  (. loop = factory.produce
8                             (LoopNode.class, t.line, t.col);
9                             .)
9 /* ... */

```

Source code 5.4: Coco/R Action code example.

Besides creating the scanner and parser component of the compiler, Coco/R is also capable of injecting Java code into the parser. We use this to construct our AST by injecting Java code from the parser depending on the different constructions encountered in the parsing process. Coco/R uses two special constructs to define how the Java code is injected. The first one is `<>` which is used to define return type and formal parameters for the method generated for the various non-terminals. `(. .)` is the second construct that specifies Java code that will be inserted at that place in the parser.

In source code 5.4 both of these constructs are used at ❶. Here the method for the non terminal $\langle\text{Loop}\rangle$ has the return type **Node** and it returns the contents of the variable *loop*. The code inside the parentheses is normal Java code and is simply inserted into the parser. The code at ❷ describes how a loop node is produced and where it is placed in the AST. The **factory.produce** method is a part of the AST component that manufactures a **Loop** node to the AST with line and column number taken from the token. How the AST component works in detail is described further in section 5.2.3 and **factory.produce** in section 5.1.2. The code generated by Coco/R from source code 5.4 is shown in source code 5.5.

```

1 Node Loop() {
2     Node loop;
3     loop = null; Node cond, s, loopBlock;
4
5     /* ... */
6
7     loop = factory.produce(LoopNode.class, t.line, t.col)
8         ;
9     /* ... */
10
11     return loop;
12 }

```

Source code 5.5: Coco/R action code inside the parser's method.

We interact with one other small part of the generated parser, namely its error component. The parser uses a fairly simple error system where a mismatch between what token it expected and what token it received increments a counter and prints a message. This counter is checked by the compiler after the source code has been scanned and parsed, and if the counter has been incremented, then the *compileStatus* boolean variable is set to false as documented in section 5.2.1.

5.2.3 Abstract syntax tree

The AST is the compiler's internal representation of the program being compiled, and is created by the scanner and parser components as documented in section 5.2.2. The AST consists of different types of nodes, all subclasses of the **Node** class. The different types of nodes and their inheritances can be seen in figure F5-4. This is done to make use of inheritance to maximize code reuse, as methods and instance variables needed in multiple different nodes are implemented in the abstract super class, so that one implementation can be used for all derived classes.

The general structure of the nodes is inspired by the format given as an example of an AST node structure in [42], and is shown in F5-5. The **Node** class is an implementation of this structure, and variations of various AST interface methods proposed in [42].

The structure is generally simple: Each node can have a number of children that can be accessed through a reference to its first child node. Accessing other children is done with the use of references to a node's sibling, a reference to the leftmost sibling in a list of children and a reference to each node's right sibling allows for iteration through a node's siblings, which the parent can utilise to access every child it might have. The parent pointer

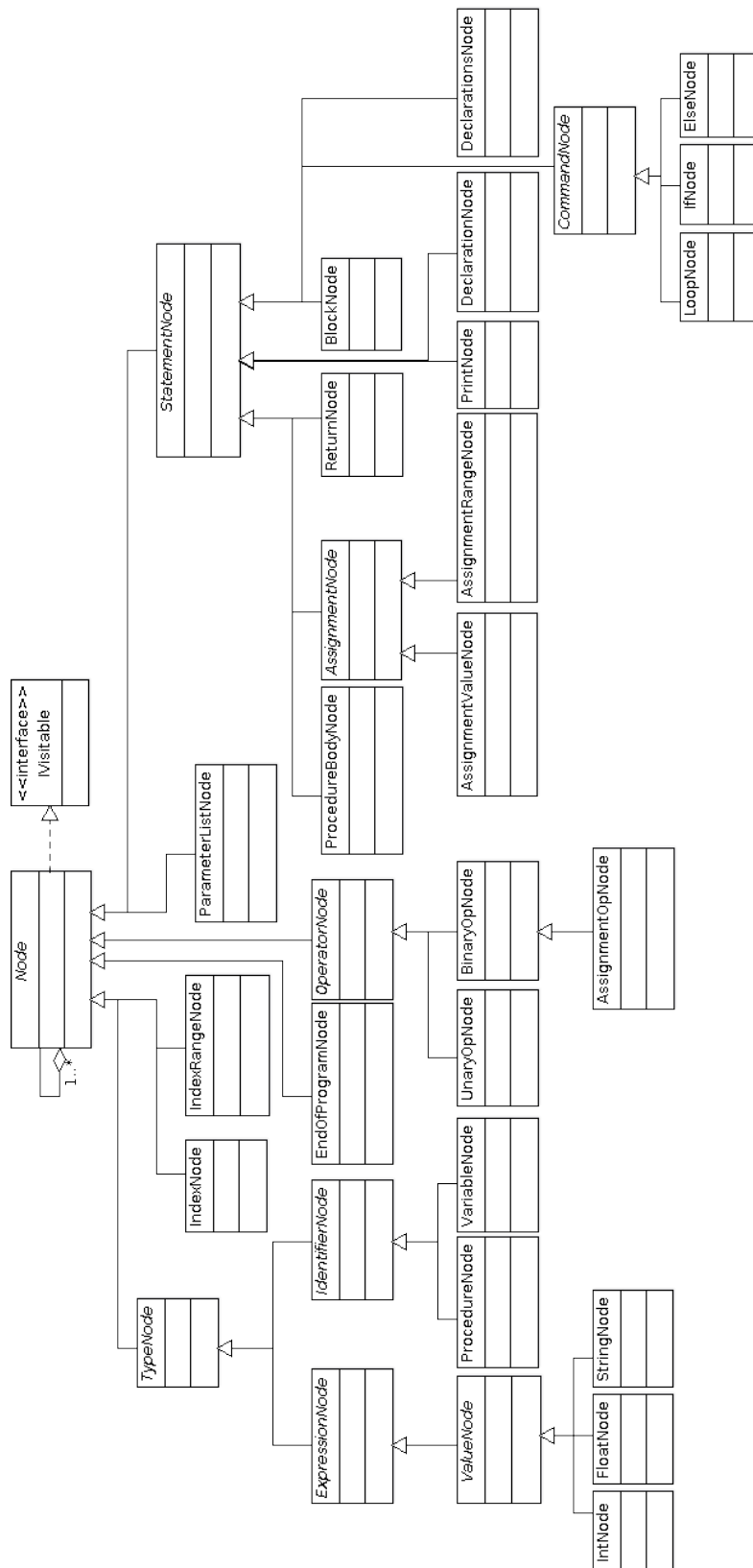


Figure F5-4: Inheritance diagram of the nodes of the AST.

on the node allows the node to ascend the tree, and visit its parent node's siblings. Having a unified system for how nodes are linked together simpli-

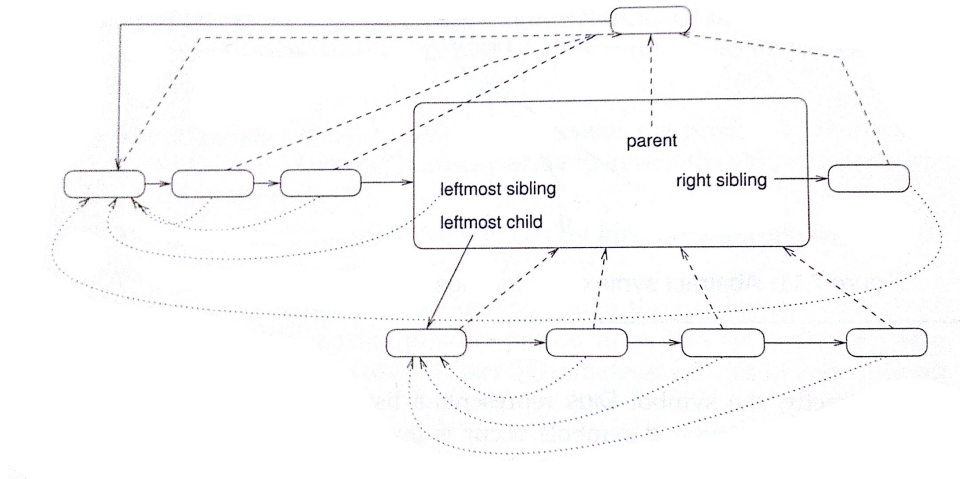


Figure F5-5: Structure of our AST nodes. [42]

fies the use of the AST as all nodes have a common interface, with a few exceptions depending on where they are placed in the inheritance hierarchy as they might contain different data sets, and removes the need for any node specific traversal method.

There is a small hack needed when a **DeclarationNode** is created with an assignment value. If a **DeclarationNode** had a assignment value with it, it meant that the **AssignmentOpNode** did not have a variable node as its left most child. This created inconsistencies while the visitors traveled the AST. To fix this a copy of the original variable node was created and adopted to an **AssignmentOpNode**. After that, the **AssignmentOpNode** adopted the **AssignmentValueNode**. The reason why a clone was created instead of assigning the original node as its child was because otherwise it would mess up the pointers to its parents and siblings. Which one should it point to when the visitors asked for its parent? That is why we decided to make a clone.

5.2.4 Symbol table

The symbol table is a component whose purpose is to hold information about all the different identifiers in the program being compiled. This simplifies later parts of the program analysis that depend heavily on identifier information like correct use of scopes and the type system.

The symbol table consists of two general constructions as shown in figure F5-6. Symbols encapsulate the various information about each identifier in the classes **VariableSymbol**, **ArraySymbol** and **ProcedureSymbol**. Environments function as a collection of symbols and define the scope they are part of. Since the various types of identifiers share information they all inherit from the abstract **Symbol** class that contains the name and type of the identifier as well as the environment that the identifier was defined in.

The instantiation of symbols are done by the **SymbolFactory** class as shown in source code 5.6 that creates the symbol with the information they need.

```

1 /* ... */
2 public static VariableSymbol produceVariable(String
    name, Datatype type, boolean isconstant,
    IEnvironment<Symbol> env) {

```

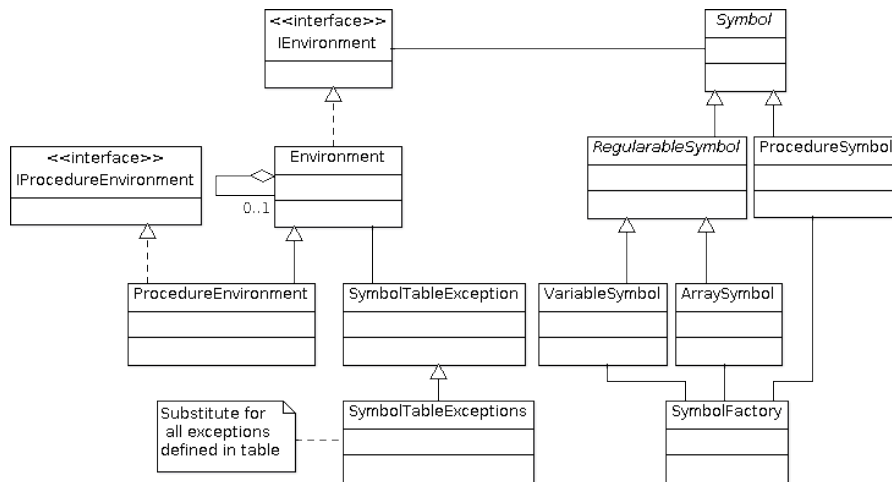


Figure F5-6: Environment and symbol classes.

```

3      return new VariableSymbol(name, type, isconstant,
4          env);
5  }

```

Source code 5.6: SymbolFactory encapsulates symbol production.

The scopes of the program is defined by generic environments which are linked by a parent pointer that points to the previous environment. A representation of this can be seen in figure F5-7. The interface of an **Environment** is defined by the **IEnvironment** interface class to ensure that it contains the methods required which allows the **Symbol** abstract class to only depend upon the interface instead of the class. The environment part is entirely separated from the symbol part as it is programmed to be generic which allows it to be instantiated as a container for all types descending from the **Object** class.

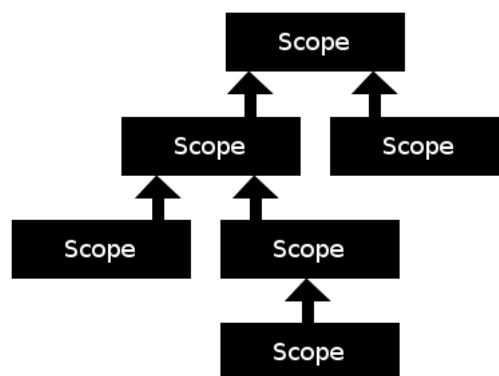


Figure F5-7: Visual interpretation of the scope system.

There are two different kinds of environments, **Environment** and **ProcedureEnvironment**, with the latter being a small extension to the former which allows the definition and retrieval of a procedure's formal parameters.

Implementation-wise the **Environment** class is just wrapping around a hash table which holds instances of the concrete **Symbol** subclasses which contains

the actual information. The class contains methods to add and retrieve elements in the hash table but also methods to control the notion of scopes. The opening of a new scope, for example when the code enters a procedure, is handled by the **openScope** method shown in source code 5.7.

```

1 public IEnvironment<TSymbol> openScope() {
2     IEnvironment<TSymbol> env = new Environment<TSymbol>(
        this, this.subscopesNum++);
3     return env;
4 }

```

Source code 5.7: The openScope method of the Environment class.

This method creates a new environment with a reference to the one currently being used and returns it to the visitors discussed in section 5.2.5 which then attaches it to the correct node. The method **get** then allows identifiers to be retrieved by going through each environment's hash table before using the parent reference as shown in 5.8.

```

1 public TSymbol get(String name) throws
    UnknownIdentifierException, NullPointerException {
2     if (this.isLocal(name))
3         return ht.get(name);
4     else if (this.getParentEnv() != null)
5         return this.getParentEnv().get(name);
6     else
7         throw new UnknownIdentifierException("Identifier
            " + name + " has not been declared.");
8 }

```

Source code 5.8: The get method of the Environment class.

This in practice implements the scope rules for CIP defined in section 4.4.2 as identifiers in the innermost scope are returned by the **get** method as they are defined in a hash table before any other instances of the same identifier. Only identifiers declared in outer scopes from the current can ever be found as the environment only refers to these. The exception shown at ❶ are together with others derivatives of the **SymbolTableException** shown in figure F5-6.

5.2.5 Visitor component

This component is the workhorse of the compiler as it contains all operations done on the AST. It is built around a visitor pattern and the reflection methods included in Java. The use of the visitor pattern allows us to define new operations for nodes in the AST without changing the node itself. This also lets us encapsulate similar methods together in a visitor, like the **TypeCheckVisitor** that has all type checking methods instead of having these methods spread out on the different AST nodes. The use of reflection allows us to specify a general visit method in the abstract visitor class and then only write the visit methods actually needed in the specific visitor. The disadvantage of this is unfortunately a performance cost. For more information about the visitor pattern itself and the use of reflection see section 5.1.3.

The visitors used in the compiler are shown in figure F5-8. They all derive from the abstract visitor class but are otherwise independent of each other. The only notable method in the abstract visitor is **visitChildren** which uses the

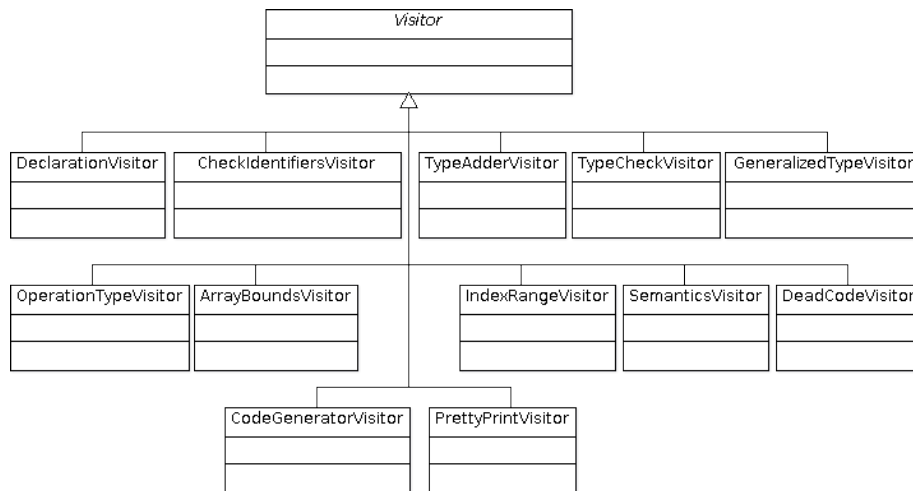


Figure F5-8: Diagram of the visitor component.

reflective **dispatch** method which finds the correct method to call based on the most specific class of the node to visit every child in the particular node with the visitor it is called in. The **visitChildren** method is shown in source code 5.9.

```

1 protected void visitChildren(Node node) {
2     if (node != null) {
3         for (Node child : node) {
4             child.accept(this);
5         }
6     }
7 }

```

Source code 5.9: The visitChildren method.

Although the method itself is fairly simple it is a good example of how the reflective visitor works. The method consists of a loop iterating through all the children of *node* with no concern for their particular type. The reflective visitor visits them by using their **accept** method which calls the **dispatch** method and sends the visitor with the method call. For more information about the **accept** method see section 5.1.3.

We have developed a collection of small visitors that all have a specific purpose. This makes the code easier to manage as the code base for each visitor is and specific to that visitor and the code base is thus sought to be kept small.

DeclarationVisitor Collects all identifiers in the form of variables and procedures and saves them in the symbol table. While it does this, scopes for procedures, if-statements, and loops are also created so all identifiers only are available in their correct context.

CheckIdentifiersVisitor Checks if the program being compiled has declared all identifiers in the scope they are being used so that CIP's scope system is used correctly. See section 4.4.2 for more documentation about the scope rules.

TypeAdderVisitor Ensures that all identifiers contains a data type so the `TypeCheckVisitor` has all the data it needs.

TypeCheckVisitor Checks if operations in the program follows CIP's type system in terms of both operations on variables and scalars. Furthermore, it also checks the return type of the procedure and the use of parameters. The type system is documented in section 4.4.1.

GeneralizeTypeVisitor This visitor generalizes types of different `TypeNodes`. If for example a program adds an `int8` to an `int32`, it changes the type of the `int8` variable to an `int32`.

OperationTypeVisitor Checks whether an operator works on arrays or scalars and stores this information.

SemanticsVisitor Does various different semantic checks. For example it checks if a **void** procedure contains any return statements which returns a type or if array indexes are out of bounds.

ArrayBoundsVisitor Checks if declarations or assignments of a range of numbers to arrays match the dimensions of the array and if possible the size of the given dimensions.

IndexRangeVisitor Checks if the indexes used in ranges matches the dimensions of the array they are used on, and that the indexes to individual elements are acceptable.

DeadCodeVisitor Analyses the program for detectable nonreachable code. The initial version only supports immutable expressions containing scalars.

CodeGenerationVisitor Defines which part of the emitter each node should use to generate code. Code generation is further described in section 5.2.7.

PrettyPrintVisitor Prints the constructed AST to standard output which was originally built for development purposes but left in the compiler as it might be useful to see the compiler's representation of the program being compiled.

5.2.6 Exception handler

The exception handler handles complications with the program during the compilation process. It does however not concern itself with exceptions happening in the code of the compiler itself, which is instead handled by Java's normal exception routines which either catch the exception and correct it during run-time, or terminate the program if the program can not recover from that particular exception. The structure of the exception handler can be seen in figure F5-9.

The exception handling mechanisms of the compiler consist of three parts. The first part is a set of errors and warnings derived from an abstract class that defines their public interface and inherited implementation from a common super class. The second part is the two factories that contain logic about how each exception should be constructed. This is based on the node type in the AST where the exception happened and what exception

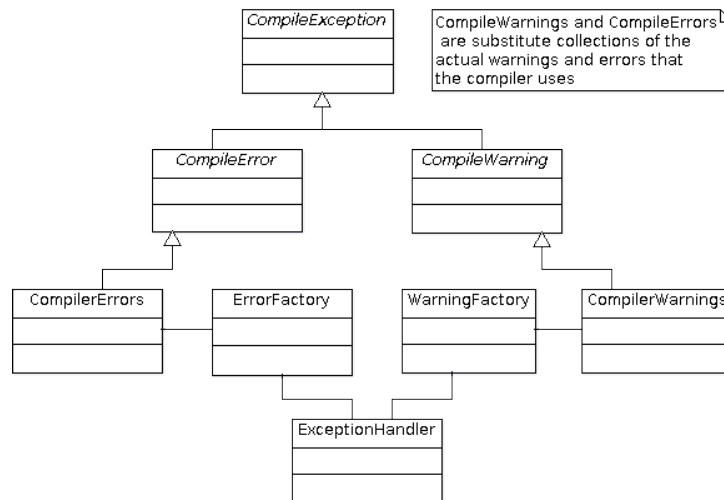


Figure F5-9: Diagram of the exception handler.

type it is. This is further described in section 5.1.2. The third and last part is the handler itself, which contains the logic needed to handle both errors and warnings. The compiler contains two separate instances of the exception handler, one that contains all errors and one that contains all warnings.

Exceptions are separated into warnings and errors because they represent problems of different severity in the program being compiled. Errors are, as the name suggests, problems that prevent the program from being compiled correctly, such as missing declarations or accessing non-existing elements in arrays. Warnings, on the other hand, are elements in the source code whose behavior may be unintended, for example code that is inaccessible due to branches.

The interfaces used by the exceptions consist mainly of two methods **getNameString** and **getInfoString**. The first method, which can be seen in source code 5.10, generates a string containing generic information shared by the different exceptions and therefore is implemented in the abstract class. The second, which generates an exception-specific information string, is abstract on the super class which forces an implementation in all errors.

```

1 protected String getNameString() {
2     return "[" + this.getClass().toString().substring(6)
        + " At Line: " + this.lineNumber + ", " + "Column
        : " + this.columnNumber + " ]\n\t\t";
3 }
  
```

Source code 5.10: The method **getNameString** for generating generic information.

The factories producing both errors and warnings are implementations of the factory pattern, which is discussed in section 5.1.2. The factory is used by calling the **factory.produce** method together with the AST node where the error happened and the type of exception that the factory should produce.

```

1 public static CompileError produce(Node node, Class<?
    extends Exception> type) {
2     CompileError ce = null;
  
```

```

3
❶ 4   if(node instanceof VariableNode) {
5       ce = produce((VariableNode) node, type);
6   }
7   /* ... */
8
❷ 9   ErrorHandler.getErrorHandler().addException(ce);
10
11   return ce;
12}

```

Source code 5.11: The error factory's **produce** method.

In source code 5.11 it can be seen that the factory checks what type of node it has received and calls the corresponding production method with the node and error type ❶. The information about the error and node type is passed on to the error which it returns. The compile error variable is then added to the error handler at ❷. The factory also returns the error to the visitor for use if it so chooses. The warning factory operates similarly but has the production routines for the different node types built into the **WarningFactory.produce** method which is only capable of producing one type of warning.

```

1 public static ErrorHandler<CompileWarning>
   getWarningHandler() {
2     if(warningInst == null)
3         ErrorHandler.warningInst = new
           ErrorHandler<CompileWarning>("warnings");
4     return warningInst;
5 }

```

Source code 5.12: Warning handler instance getter.

The exception handler's interface for the handler consists of two getter methods, one for the error handler **getErrorHandler** and one for the warning handler **getWarningHandler**. This is done to enhance encapsulation and allow the use of the singleton pattern documented in section 5.1.1. This pattern prevents the instantiation of more than one of each handler to ensure that all exceptions are saved at the same place, as seen in source code 5.12. The main purpose of the handler is to aggregate the exceptions from the factory and therefore contain both **add** and **remove** methods. The handler also contains a print method called **printExceptions** which prints all exceptions that the handler currently contains to the standard output.

5.2.7 Code generator

This section contains information about the code generation part of the compiler. The code generator consists of three parts where the first one is the code generator visitor class which provides information for the later parts in the form of code fragment objects. These objects are made by the code fragment generator class which gets its information for the objects from the code generator visitor while it traverses the AST. The code fragment generator is a wrapper around the string buffer class to simplify the process of transforming the AST information to C code with inline assembler by supplying the code generator visitor class with methods to ease this. Code

fragment objects are then used by the code emitter class to output the objects as a string to the specified output file.

5.2.7.1 Code generator visitor

The heart of the code generator is simply another visitor. It generates target code directly off the AST. However it differs a bit from the other visitors since it distributes a few responsibilities to other objects. The **CodeGeneratorVisitor** class does not handle the generation of the strings containing code, or *code fragments* as we call them. The visitor leaves this responsibility to the **CodeFragmentGenerator** class which knows all about how to generate code fragments and concatenate them, thus hiding the details of how it is done. Through the public API of **CodeFragmentGenerator** the **CodeGeneratorVisitor** is capable of requesting various code fragments. The **CodeFragmentGenerator** does not know how to emit code, this responsibility is left to the **CodeEmitter** class.

For instance when the **CodeGeneratorVisitor** visits a **BinaryNode** the visitor has to determine whether it is a complex binary operation, one that involves operations on arrays, or a primitive operation that involves only primitive types.

```

1 public Object visit(BinaryOpNode node) {
2     if (node.getOperationType() == OperationType.Complex)
3         return this.visitComplexBinaryOperation(node);
4     else
5         return this.visitPrimitiveBinaryOperation(node);
6 }

```

Source code 5.13: The code generator visitor calls **visit** on **BinaryOpNode**.

The operation type is determined at ❶, the operation type has already been resolved by another visitor **OperationTypeVisitor**.

```

1 private CodeFragment visitPrimitiveBinaryOperation(
2     BinaryOpNode node) {
3     CodeFragment cf = null;
4     // Visit left operand
5     CodeFragment operand1 = (CodeFragment)node.
6     getLeftMostChild().accept(this);
7     // Visit right operand
8     CodeFragment operand2 = (CodeFragment)node.
9     getLeftMostChild().getRightSibling().accept(this);
10    // Generate CodeFragment
11    cf = this.getCodeFragmentGenerator().
12    createBinaryOperation(node.getOperator(), operand1,
13    operand2);
14    return cf;
15 }

```

Source code 5.14: The code generator calls **visitPrimitiveBinaryOperation** on **BinaryOpNode**.

Source code 5.14 shows how the visitor handles the primitive binary operation. The method is straight forward:

1. Visit the left operand ❶ to obtain its code fragment.
2. Visit the right operand ❷ to obtain its code fragment.

3. Let the **CodeFragmentGenerator** ❸ generate a valid code fragment that concatenates the two previous fragments.

The resulting code fragment, which is a binary operation in this example, is then propagated further up the AST. The method basically performs a postorder traversal of the subtree. The propagated code fragment *cf* is caught and emitted in another node. For instance in this example it may only be emitted when visiting an assignment statement. Source code 5.15 shows the logic handling code generation for assignments.

```

1 /*
2  * @desc Generates code for an assignment operation
3  */
4 public Object visit(AssignmentOpNode node) {
❶ 5   if (node.getOperationType() == OperationType.Complex)
6       this.visitComplexBinaryOperation(node);
7   else {
8       // Let BinaryOpNode visit-method handling
          CodeFragment generation
❷ 9       CodeFragment assign = (CodeFragment)this.visit(((
          BinaryOpNode)node));
10      // Generate statement
11      CodeFragment cf = this.getCodeFragmentGenerator().
          createState(assign);
12      // Emit codefragment
❸ 13      getEmitter().emitln(cf);
14  }
15  return null;
16 }

```

Source code 5.15: Assignment statement handling and code emitting

Since an **AssignmentOpNode** is a **BinaryOpNode** it has to deal with the same case, it has to determine whether the current operation involves array operations ❶. If it does it calls a specialised method suitable to handle such expressions. However if the operation type is primitive then the visitor falls back on the **visit** method for **BinaryOpNode** and lets it obtain the proper code fragment for the expression. When the visitor returns to this node again it emits the obtained code fragment by issuing a call to the **emitln** method ❸ on the **CodeEmitter**.

```

1 private CodeFragment visitComplexBinaryOperation(
    BinaryOpNode node) {
❶ 2   CodeFragment leftOperand = (CodeFragment)node.
        getLeftMostChild().accept(this);
❷ 3   CodeFragment rightOperand = (CodeFragment)node.
        getLeftMostChild().getRightSibling().accept(this);
4 /* ... */
❸ 5   CodeFragment rangeExp = this.getArrayOperationRange(
        node.getLeftMostChild());
6 /* ... */

```

Source code 5.16: The code generator calls **visitComplexBinaryOperation** on **BinaryOpNode**.

The source code in 5.16 shows how the visitor handles complex binary operations. The first part of the code is also straight forward:

- Visit the left operand ❶ to obtain its code fragment.

- Visit the right operand ❷ to obtain its code fragment.
- Gets the range to operate on from the left child, as we are certain that this is the array type because of the way the AST is build ❸.

```

1 /* ... */
❶ 2     CodeFragment leftPtr = this.
   getCodeFragmentGenerator().createTemporaryVariable(
   true);
3     CodeFragment rightPtr;
❷ 4     this.emitTmpVarDeclaration(type, 1, leftPtr, this.
   getCodeFragmentGenerator().createAddressOf(leftOperand
   ));
❸ 5     if (this.retTmpArray == true) {
6         rightPtr = rightOperand;
7         resultArray = rightPtr;
8     } else {
9         rightPtr = this.getCodeFragmentGenerator().
   createTemporaryVariable(true);
10        this.emitTmpVarDeclaration(type, 1, rightPtr,
   this.getCodeFragmentGenerator().
   createAddressOf(rightOperand));
11        resultArray = this.emitResultArrayAllocation(type
   , elements);
12        this.retTmpArray = true;
13    }
14 /* ... */

```

Source code 5.17: The code generator calls **visitComplexBinaryOperation** on **BinaryOpNode**. continued.

The continued source code in 5.17 shows logic handling complex operations on arrays when both operands are arrays. In the beginning of the method call code fragments are obtained for both *leftOperand* and *rightOperand*. Since both operand are arrays we will be using to pointers to point to the beginning of the operation range in both arrays. One may note that the *leftOperand* code fragment contains the left operand array identifier associated with the proper index dereferences. The *leftPtr* points to the left operand ❶. At ❷ the *leftPtr* is emitted and initialized to the address of left operand. The *rightPtr* is not initialized and emit until later.

The variable *retTmpArray* is an instance variable used to determine whether the result array has already been allocated for the expression ❸. This is a little bit of a hack which has been imposed due to inconsistent design of the **CodeFragment**-class, we need some kind of information describing what returns from the operand visits so that we may keep reusing an already allocated result array for the entire expression. *retTmpArray* simply serves as a flag indicating whether any of the operands returned is this special temporary result array. On the basis of this information the *rightPtr* is either assigned to point to the result array or it points to the address of the right operand array.

```

1 /* ... */
❶ 2     SSE2ASMCode = this.getCodeFragmentGenerator().
   createSSE2Operation(node.getOperator(), resultArray,
   leftPtr, rightPtr, SSE2elements, type, false);

```



```

② 3   leftOversHandle = this.getCodeFragmentGenerator().
    createSSE2LeftOversHandle(node.getOperator(),
    resultArray, leftPtr, rightPtr, SSE2elements, elements
    , false);
4/* ... */
5}

```

Source code 5.18: The code generator calls **visitComplexBinaryOperation** on **BinaryOpNode**. continued.

Finally in source code 5.18 the **CodeFragmentGenerator** generates ASM C inline code fragments objects encapsulating the SSE2 instructions ❶. The *leftOversHandle* contains a code fragment to handle any elements from the arrays that could not fill out a **XMM**-register.

One may notice that since the result array is propagated further up AST until a copy instruction operation occurs. However since we do not have any optimisations in our compiler at the moment the following expression imposes an unnecessary overhead:

```

1 // Let A and B be arbitrary arrays
2 // Let i and j be arbitrary integers
3 A[i][:j] = A[i][:j] + B[i][:j];

```

Source code 5.19: Overhead imposed due to lack of optimisations

In source code 5.19 the result of the plus-operation is stored in a temporary result array just to be copied right back into the array *A*. Instead it would have been smart if our compiler would recognize that source and destination were the same, thereby skipping one allocation and copy operation.

5.2.7.2 Code fragment generator

We will show a simple example of a method from the **CodeFragmentGenerator**-class. Every single method of the class is just building proper code fragments. The source code 5.20 shows the code generation a code fragment for a binary operation.

```

1 public CodeFragment createBinaryOperation(Operator op,
    CodeFragment operand1, CodeFragment operand2) {
2   CodeFragment cf = this.createCodeFragment();
① 3   cf.appendCode(operand1, false); cf.appendCode(Utils.
    convertOperatorToString(op)); cf.appendCode(operand2);
② 4   cf = this.wrapParenthesesAround(cf);
5   return cf;
6}

```

Source code 5.20: The code fragment generators **emit**.

At ❶ the operands are correctly connected to their operator and then returned to the caller. One may notice that we put parentheses around every expression ❷ we do this to ensure that the precedence rules of our programming language is preserved in the target language.

5.2.7.3 Code emitter

The purpose of the code emitter class is to provide methods to print the code fragment objects into the specified output file. It handles some primitive pretty printing as well by asking the boolean variable *atNewLine* whether this

is the first string to be printed on the current line. If true, it prepends the level of indentation (some arbitrary number of whitespaces) to the string. The level of indentation is increased and decreased in each visit of a **BlockNode** and **ProcedureBodyNode**.

```
1 public void emit(String s) {  
2     if (this.atNewLine) {  
3         s = this.getIndentationAsString() + s;  
4         this.atNewLine = false;  
5     }  
6     this.getStream().print(s);  
7 }  
8  
❶ 9 public void emit(CodeFragment cf) {  
10     this.emit(cf.toString());  
11 }
```

Source code 5.21: The code fragment generators **emit**.

In source code 5.21 we can see that if a code fragment object is passed as a parameter the object is converted to a string ❷. This string is then printed in the specified output stream that **getStream** has information about.

5.2.7.4 Memory management

Since all arrays in our language are heap allocated we need some kind of memory management. However constructing and implementing a full-fledged garbage collector is out of the scope of this project. Therefore we have chosen to use a simple stack-based allocation framework written in C called **cipalloclib**. Every array is allocated through this framework. A reference to each allocation is pushed onto a global storage class stack. Right before the program terminates a call to **cip_free()** is issued which pops and frees each element from stack. This give us a very simple memory management.

Test of compiler

In the following chapter will we describe the way in which we have tested our compiler, as well as the results we have obtained from the tests.

There are several different reasons for testing our compiler. The first is that we need to ensure that our compiler produces the correct code so that the produced code is equivalent to the intended semantics. As the purpose of this project was to create a language that could effectively use SIMD instructions, we must ensure that the SIMD instructions are generated correctly.

Since our language is concerned with execution speed through the use of the SSE instruction set that in theory is faster than sequential instructions, it is necessary to compare what our compiler generates performance-wise to what other, well-established compilers such as the GCC compiler generates.

6.1 Testing Methods

To test our compiler we will create several test programs that will remain unchanged throughout our testing phase so that there is always a fixed constant in our tests. By translating these programs into C it is possible to compare the result of the C program and the CIP program and see how they differ. This has been done regularly throughout the development process. The sample programs we have created can be seen in appendix B.

The only explicit test we will create is a performance test. We will test a performance-heavy program that allocates multiple arrays which we will use to test the performance of the generated program by our compiler. These results are going to be compared to the performance of an equivalent program in C and compiled with GCC both unoptimised and optimised.

As performance can vary between different CPUs we intend to execute the performance benchmark using the different CPUs we have available and compare the results.

6.2 CIP Benchmark

The benchmarking between C and CIP is done by running equivalent programs with small arithmetic operations on large amounts of data. This is done in practice by allocating two arrays used as the right and left operand of an arithmetic operation and then using a loop in C or CIP to perform the operation. As the CIP compiler always uses a third array to save the result of array operations and then move the elements to the correct place, two different tests have been written for C: A direct test where the result

is saved directly to the target array, and an indirect test where it, as in the CIP program, is moved after the operation is complete. All arrays in the programs are allocated and deallocated using **cipalloclib** which is described in section 5.2.7.

We have used two different computers in the performance benchmark: One with an Intel Core Duo 2 with 2 GB of RAM, and another with an Intel Core i7 and 8 GB of RAM, both running a 64-bit version of Linux. These were used to run two tests, one doing additions between two arrays and another which does multiplication. Using addition as the test instruction was done because it is available as a single instruction for all packed data types, which allows us to test the performance of the programs while only changing the data type used.

	1.5 GB allocated		3 GB allocated		4.5 GB allocated	
data type	C	CIP	C	CIP	C	CIP
8-bit Integer	6.410	4.495	12.874	8.981	19.213	13.460
16-bit Integer	3.567	2.520	7.314	5.033	10.686	7.549
32-bit Integer	1.826	1.493	3.647	2.983	5.472	4.475
64-bit Integer	1.026	0.985	2.051	1.984	3.073	2.949
32-bit Floating	1.604	1.268	3.199	2.529	4.796	3.791
64-bit Floating	1.077	1.034	2.156	2.066	3.073	3.097

Table T6-1: Direct array additions using an Intel Core i7.

	1.5 GB allocated		3 GB allocated		4.5 GB allocated	
data type	C	CIP	C	CIP	C	CIP
8-bit Integer	8.224	4.492	16.446	8.978	24.666	13.463
16-bit Integer	4.581	2.519	9.151	5.030	13.722	7.545
32-bit Integer	2.397	1.493	4.789	2.985	7.176	4.475
64-bit Integer	1.335	0.968	2.670	1.966	3.998	2.952
32-bit Floating	2.166	1.272	4.331	2.532	6.499	3.861
64-bit Floating	1.387	0.986	2.767	1.966	4.148	2.952

Table T6-2: Indirect array additions using an Intel Core i7.

	1.5 GB allocated	
data type	C	CIP
8-bit Integer	8.955	6.855
16-bit Integer	4.986	4.617
32-bit Integer	2.976	3.535
64-bit Integer	2.223	1.925
32-bit Floating	3.721	4.368
64-bit Floating	2.219	3.232

Table T6-3: Direct array additions using an Intel Core 2 Duo.

The other test is done using SSE instructions to multiply array elements with each other. This is done to compare the use of a simple instruction in the form of addition with the more complex instruction in the form of multiplication without changing the data type. Since multiplication only exists

	1.5 GB allocated	
data type	C	CIP
8-bit Integer	12.223	6.490
16-bit Integer	6.308	4.508
32-bit Integer	4.069	3.532
64-bit Integer	3.142	3.167
32-bit Floating	4.827	4.270
64-bit Floating	3.121	3.161

Table T6-4: Indirect array additions using an Intel Core 2 Duo.

as an instruction for floating point data types, this test is only conducted on these.

	1.5 GB allocated		3 GB allocated		4.5 GB allocated	
data type	C	CIP	C	CIP	C	CIP
32-bit Floating	1.603	1.264	3.193	2.535	4.799	3.222
64-bit Floating	1.077	1.035	2.151	2.067	3.222	3.096

Table T6-5: Direct array multiplications using an Intel Core i7.

	1.5 GB allocated		3 GB allocated		4.5 GB allocated	
data type	C	CIP	C	CIP	C	CIP
32-bit Floating	2.174	1.035	4.328	2.541	6.492	3.788
64-bit Floating	1.390	1.035	2.771	2.065	4.156	3.096

Table T6-6: Indirect array multiplications using an Intel Core i7.

	1.5 GB allocated	
data type	C	CIP
32-bit Floating	4.248	4.248
64-bit Floating	2.222	3.164

Table T6-7: Direct array multiplications using an Intel Core 2 Duo.

	1.5 GB allocated	
data type	C	CIP
32-bit Floating	4.819	4.248
64-bit Floating	3.104	3.164

Table T6-8: Indirect array multiplications using an Intel Core 2 Duo.

The results of this test can be seen in the above table T6-1, table T6-2, table T6-5, table T6-6, table T6-3, table T6-4, and table T6-7 and it can be seen that CIP is faster than C when GCC is compiling without any optimization, and that the performance leap is most significant when smaller data types are used as CIP is capable of computing larger amounts of elements in parallel.

The performance lead was quickly lost when GCC's optimisations were enabled which allowed the compiler to use optimisation functions not in-

cluded in our compiler, as we consider them out of scope of this project. So the performance gained by using CIP's data parallelism gives it an edge when compared to another compiler without optimisations like itself, but is outmatched when compared to the optimisation routines of a modern compiler.

Conclusion and discussion

In this chapter we will conclude upon our project and discuss issues that came up during the project process as well as reflect upon ways in which to improve and expand upon the language and compiler presented in this report.

7.1 Conclusion

This section will contain the conclusion of the whole project. The analysis in section 2 helped with the understanding of SIMD and gave inspiration to the language creation process which resulted in CIP. With this knowledge that was gained through the analysis we were able to make important design restrictions to the compiler and therefore the language which is documented in section 3. These restrictions and the understanding of SIMD helped give purpose, further restrictions and helped form the language design in section 4 into the current which focuses on computing arrays efficiently with data parallelism. The language's formal and informal grammar, syntax and semantics, operation rules and contextual rules that were made in the language design section helped us with choices and overall design of the compiler which is located in the compiler architecture section 5.

We formalised our programming language by creating a CFG expressed using the Extended Backus-Naur Formalism (EBNF). This imposes a natural frame in how one may express oneself in our language. However by enforcing constraints on the language through EBNF, it is possible to have a consistent language construction, which is very important when constructing the abstract syntax tree. For more information on the design decisions behind the language, see section 4.

Since this is the first version of our programming language we have chosen only to support a predefined handful of data types. We wanted to create a general purpose programming language with some special array operations, therefore the supported integer and floating-point types are sufficient to fulfill this purpose. We made arrays nearly first class citizens of our programming language, however they fail in being first class due to returning arrays from procedures being unsupported at the moment and operations on arrays requiring the programmer to specify ranges on arrays.

The benchmarks of our language showed that we could outperform GCC's C compiler or at least have comparable performance in most of the tests. That said, the result was obtained by having no optimisations whatsoever. However GCC outperformed our language *greatly* when using optimisation flags (-O3). This result was expected, but we are surprised that the code generated by our compiler executes faster than the code generated by GCC

without optimisation flags. Especially since we have an extra overhead due to the copy instruction and the temporary result container array allocation in each statement.

7.2 Discussion

This section contains the discussion about features that could have been implemented but were left out from this version of our language due to decisions made during the development process. The reasons for not including these in the final system vary from early implementation limits to them being out of scope. We will also discuss why we would like to include these features in a future version of our language and compiler.

7.2.1 Expansion of the AST

During the development process we found that the way in which we had structured our AST gave some issues during implementation. We tried in our AST to have as few node classes as possible to reduce complexity, but we found during implementation that we would have benefitted from additional node classes that could contain additional information about constructs in the language during compilation. Therefore we would like an expansion of the AST that would include new nodes on the tree if we were to continue developing the compiler. For example we would like to implement an array node in the tree that would let us store information about arrays in a more natural way than we are currently doing.

7.2.2 Intermediate format

Instruction selection is often simplified by translating source language constructs into a form more suitable for code improving transformations as a step before code generation. We would like to perform optimisation on the code that we generate in order to obtain better performance. In order to facilitate this, we would like to introduce an intermediate format for our compiler that is better suited for optimisation purposes. An intermediate language is typically more concise and abstract than the lower level target. This allows the developer to focus on the code generation process without having to think about the underlying details of the machine's instruction set. This intermediate format will make it possible to optimise and make the code platform more independent. Some examples of intermediate languages is three address code and static single assignment. [42]

7.2.3 User-friendly features

The language presented in this report does not offer that many user-friendly code constructs in the language compared to other mainstream languages such as C and Java. We would like implement such constructs as the `++` or `--` operators which would make writing the code more efficient, as the developer would have to write less code. Other things we could implement in the future would be the ability to make for-loops which many people are accustomed to. The string data type is not implemented in CIP but could be implemented and could greatly benefit the developer in everything from debugging to general programming. We would also like to allow the print

statement to work one one and two-dimensional arrays, so that it prints these out in an easy-to-read format. The ability to print more than one element to standard output at a time would also be beneficial and if the string data type is implemented a way to combine variables, arrays and strings in a print would increase writeability as well. Another useful feature would be to allow the program to accept some kind of input, perhaps like the `scanf` function in C, which could be implemented in the future.

7.2.4 Array operations

We would have liked to have more array operations available in CIP that would let the developer select more than rows of arrays. This was not implemented because it was out of scope for this report as the system would need a runtime environment to take care of all the array information or a restructuring of the AST because our current AST did not contain this information.

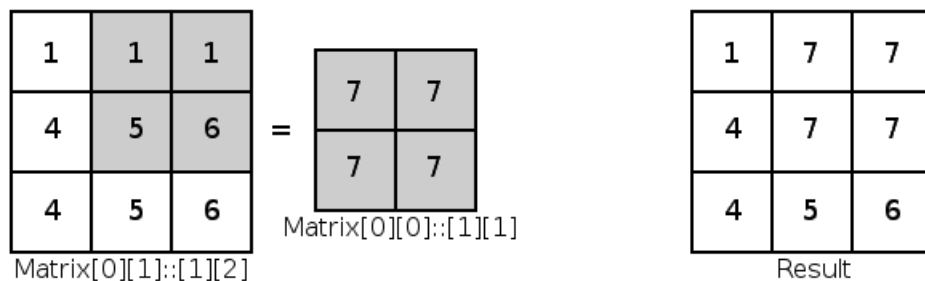


Figure F7-1: Illustration of box selection of array and assignment.

We would have liked to have the selection available for the developer in a way so they could make array selection across rows. The implementation we thought of was a box selection as seen in figure F7-1. With this box selection it would be possible to select column representations as well as row selections and a smaller matrix selection in our array as seen in F7-2. It would even be possible to select subarrays of arrays with an arbitrary number of dimensions.

Something to take into account though is whether or not that operation is even worth it. Since all arrays in our language are represented as single dimension arrays in C, it would mean that special calculations would be necessary in order to get the elements in those positions when operating with columns. Not only that, we would have to put all of the elements into a temporary array and apply the operation to it. Finally we would have to insert the elements back into the original array in the correct positions. That is a lot of overhead just to be able use utilize SSE2 instructions on columns.

7.2.5 Blocks of operations

The ability to create blocks of code which would utilize SIMD instructions would also have been a nice addition to CIP. This would open up the ability to define certain variables that would have the same operation applied to them at the same time through the use of SSE2 instructions. The problem

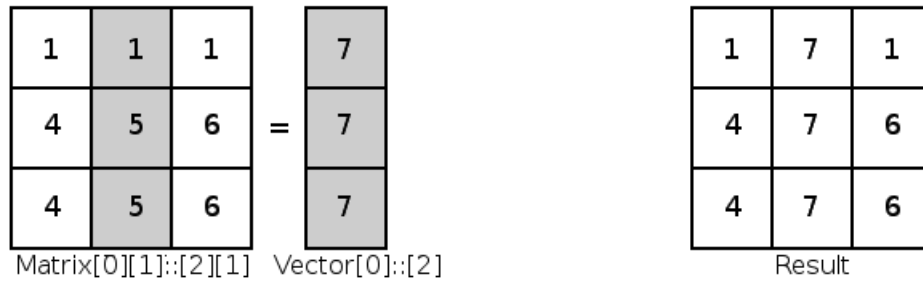


Figure F7-2: Illustration of vertical selection of array and assignment.

with it though was that it would require a lot more analysis and a different way of traveling the nodes of the AST representing the block.

7.2.6 Runtime environment

A runtime environment would allow for more dynamic solutions across the whole system, from better overflow checks to saving array sizes to making out-of-bounds exceptions. This would open up opportunities for the system to develop better debugging tools. The downside is that it would introduce an extra layer of software which would take up some additional system resources.

7.2.7 Evolutionary approach

While our working process could be considered a mixture of the evolutionary approach with the constructive approach, it was mainly focused around the evolutionary approach. Halfway through our project, we decided to reconstruct our compiler from scratch. At that point we decided it would be beneficial to model it properly so that everyone had a clear idea of what our compiler should look like 5.2. That was a huge benefit and greatly simplified our working process. In retrospect, this is probably something we should have done in the beginning. Something to remember though is that we did not have all the knowledge needed to properly model a compiler at the beginning of the project. This is something that we can and should do next time we work on a compiler.

Appendices

Context-Free Grammar

A context-free grammar documenting the syntax of the programming language CIP is presented here, written in Extended Backus-Naur notation [31]. Non-terminal symbols are enclosed in $\langle \rangle$, and terminal symbols are written with a **Bold** font.

A.1 The grammar

```

 $\langle \text{CIP} \rangle \rightarrow \langle \text{Statements} \rangle$ 
 $\langle \text{Statements} \rangle \rightarrow \langle \text{Statement} \rangle \{ \langle \text{Statement} \rangle \}$ 
 $\langle \text{Statement} \rangle \rightarrow ( ( \langle \text{Assignment} \rangle \mid \langle \text{Print} \rangle \mid \langle \text{Declarations} \rangle \mid \langle \text{ProcedureReturn} \rangle ) ;$ 
 $\mid \langle \text{Command} \rangle \mid \langle \text{Procedure} \rangle$ 

 $\langle \text{Declarations} \rangle \rightarrow [ \text{const} ] \langle \text{Datatype} \rangle \langle \text{Declaration} \rangle \{ , \langle \text{Declaration} \rangle \}$ 
 $\langle \text{Declaration} \rangle \rightarrow \langle \text{Var} \rangle [ = \langle \text{AssignmentValue} \rangle ]$ 

 $\langle \text{Procedure} \rangle \rightarrow \text{procedure} ( \langle \text{Datatype} \rangle \mid \text{void} ) \text{id} \langle \text{FormalParameters} \rangle \langle \text{Statements} \rangle$ 
 $\text{end}$ 
 $\langle \text{FormalParameters} \rangle \rightarrow ( [ \langle \text{FormalParameter} \rangle \{ , \langle \text{FormalParameter} \rangle \} ] )$ 
 $\langle \text{FormalParameter} \rangle \rightarrow \langle \text{Datatype} \rangle [ ] \text{id}$ 
 $\langle \text{ProcedureReturn} \rangle \rightarrow \text{return} \langle \text{Expressions} \rangle$ 

 $\langle \text{Datatype} \rangle \rightarrow \text{Int} \mid \text{Float}$ 
 $\langle \text{Int} \rangle \rightarrow \text{int8} \mid \text{int16} \mid ( \text{int} \mid \text{int32} ) \mid \text{int64}$ 
 $\langle \text{Float} \rangle \rightarrow ( \text{float} \mid \text{float32} ) \mid \text{float64}$ 

 $\langle \text{Var} \rangle \rightarrow \text{id} [ \langle \text{Index} \rangle [ \langle \text{IndexRange} \rangle ] \mid \langle \text{ActualParameters} \rangle ]$ 
 $\langle \text{ActualParameters} \rangle \rightarrow ( [ \langle \text{Expressions} \rangle \{ , \langle \text{Expressions} \rangle \} )$ 
 $\langle \text{Index} \rangle \rightarrow [ \langle \text{Expressions} \rangle ] \{ [ \langle \text{Expressions} \rangle ] \}$ 
 $\langle \text{IndexRange} \rangle \rightarrow :: [ \langle \text{Expressions} \rangle ]$ 

 $\langle \text{Assignment} \rangle \rightarrow \text{Var} [ = \langle \text{AssignmentValue} \rangle ]$ 
 $\langle \text{AssignmentValue} \rangle \rightarrow \langle \text{Expressions} \rangle \mid \langle \text{AssignmentRange} \rangle$ 
 $\langle \text{AssignmentRange} \rangle \rightarrow \{ \langle \text{AssignmentValue} \rangle \{ , \langle \text{AssignmentValue} \rangle \} \}$ 

 $\langle \text{Expressions} \rangle \rightarrow \langle \text{LogicalOr} \rangle$ 
 $\langle \text{Expression} \rangle \rightarrow ( \langle \text{Expressions} \rangle ) \mid \langle \text{Value} \rangle$ 
 $\langle \text{Not} \rangle \rightarrow \langle \text{Expression} \rangle \mid - \langle \text{Expression} \rangle \mid ! \langle \text{Expression} \rangle$ 
 $\langle \text{MultiplyDivide} \rangle \rightarrow \langle \text{Not} \rangle \{ ( * \mid / \mid \% ) \langle \text{Not} \rangle \}$ 
 $\langle \text{AddMinus} \rangle \rightarrow \langle \text{MultiplyDivide} \rangle \{ ( + \mid - ) \langle \text{MultiplyDivide} \rangle \}$ 
 $\langle \text{GreaterLesser} \rangle \rightarrow \langle \text{AddMinus} \rangle \{ ( < \mid \leq \mid > \mid \geq ) \langle \text{AddMinus} \rangle \}$ 
 $\langle \text{EqualorNot} \rangle \rightarrow \langle \text{GreaterLesser} \rangle \{ ( == \mid != ) \langle \text{GreaterLesser} \rangle \}$ 
 $\langle \text{LogicalAnd} \rangle \rightarrow \langle \text{EqualorNot} \rangle \{ \&\& \langle \text{EqualorNot} \rangle \}$ 
 $\langle \text{LogicalOr} \rangle \rightarrow \langle \text{LogicalAnd} \rangle \{ \mid \mid \langle \text{LogicalAnd} \rangle \}$ 

 $\langle \text{Print} \rangle \rightarrow ( \text{print} \mid \text{println} ) ( \text{Value} \langle \text{out val} \rangle \mid \text{string} )$ 
 $\langle \text{Value} \rangle \rightarrow \text{inum} \mid \text{fnum} \mid \langle \text{Var} \rangle$ 

 $\langle \text{Command} \rangle \rightarrow \langle \text{Loop} \rangle \mid \langle \text{If} \rangle$ 
 $\langle \text{Loop} \rangle \rightarrow \text{while} ( \langle \text{Expressions} \rangle ) [ \langle \text{Statements} \rangle ] \text{end}$ 
 $\langle \text{If} \rangle \rightarrow \text{if} ( \langle \text{Expressions} \rangle ) \langle \text{Statements} \rangle [ \text{else} \langle \text{Statements} \rangle ] \text{end}$ 

```


Cip example programs

This section contains example programs written in the programming language CIP. The programs are used to showcase the various features that the language supports and demonstrates how to implement known algorithms.

B.1 Hello world

```
1//Prints hello world
2print "Hello World\n";
3
4//Print with a implicit newline
5println "Hello World";
```

Source code B.1: Hello world in CIP.

B.2 Cross product

```
1/*
2 * Calculates the cross product of two vectors.
3 */
4
5 procedure void CrossProduct(int8[] A, int8[] B, int8[] C
6 )
7     // 2 * 6 - 3 * 5 = 12 - 15 = -3
8     C[0] = (A[1] * B[2]) - (A[2] * B[1]);
9
10    // 3 * 4 - 1 * 6 = 12 - 6 = 6
11    C[1] = (A[2] * B[0]) - (A[0] * B[2]);
12
13    // 1 * 5 - 2 * 4 = 5 - 8 = -3
14    C[2] = (A[0] * B[1]) - (A[1] * B[0]);
15
16end
17
18int8 Array1[3] = {1,2,3}, Array2[3] = {4,5,6}, Array3[3]
19    = {0,0,0};
20//returns array[3] = {-3,6,-3}.
21CrossProduct(Array1, Array2, Array3);
22
23//Prints the result
24print Array3[0]; print Array3[1]; print Array3[2];
```

Source code B.2: Computation of the cross product in CIP.

B.3 Fibonacci

```
1/* This program will find all the even Fibonacci
2 * numbers that are less than four million
3 * and print the sum of all these.
4*/
5
6int fib1 = 1, fib2 = 2;
7int result = 0;
8
9procedure int fibonacci (int fib1, int fib2, int result)
10  if(fib2 > 4000000)
11    return result;
12  else
13    if(fib2 % 2 == 0)
14      result = result + fib2;
15    end
16    return fibonacci(fib2, (fib1 + fib2), result);
17  end
18end
19
20println "The sum of all even-valued fibonacci numbers
21        with a value below four million is:";
22println fibonacci(fib1, fib2, result);
```

Source code B.3: Computation of Fibonacci numbers in CIP.

B.4 Euclidean

```
1/*
2 * Three different implementations of Euclid's algorithm.
3 * One based on addition, one based on subtraction, and
4 * one based on recursion.
5 * http://en.wikipedia.org/wiki/Euclidean_algorithm#
6 * Implementations
7*/
8
9//Procedure declarations
10procedure int gcdAddition(int a, int b)
11
12  int t;
13
14  while (b != 0)
15    t = b;
16    b = a % b;
17    a = t;
18  end
19
20  return a;
21end
22
```



```
21 procedure int gcdSubtraction(int a, int b)
22
23     if (a == 0)
24         return b;
25     end
26
27     while (b != 0)
28         if (a > b)
29             a = a - b;
30         else
31             b = b - a;
32         end
33     end
34
35     return a;
36 end
37
38 procedure int gcdRecursive(int a, int b)
39
40     if (b == 0)
41         return a;
42     else
43         return gcdRecursive(b, a % b);
44     end
45 end
46
47 //Main program
48 int x, y;
49 int gcda, gcds, gcdr;
50
51 x = 123; y = 481;
52
53 gcda = gcdAddition(x,y);
54 gcds = gcdSubtraction(x,y);
55 gcdr = gcdRecursive(x,y);
56
57 println "Input was:";
58 print x; print " "; print y;
59
60 //Multiple newlines for pretty formatting
61 print "\n\n";
62
63 println "The procedures made the following result";
64 print gcda; print " "; print gcds; print " "; print gcdr;
65
66 print "\n";
```

Source code B.4: Computation of greatest common divisor in CIP.

B.5 Pythagorean triplets

```
1 procedure int64 calculate()
2     int64 a = 0;
3     int64 b = 0;
4     int64 c = 0;
5
6     while (c <= 1000)
```

```

7   while (b <= c)
8       while (a <= b)
9           if((a < b) && (b < c) && ((a*a) + (b*b) == (c*c))
              && ((a + b + c) == 1000))
10              return (a * b * c);
11          end
12          a = a + 1;
13      end
14      b = b + 1;
15      a = 0;
16  end
17  c = c + 1;
18  b = 0;
19  end
20
21  return 0;
22 end
23
24 println "The product of the only pythagorean triplet for
           which a + b + c = 1000 is:";
25 println calculate();

```

Source code B.5: Computes pythagorean triplets in CIP.

B.6 Matrix manipulation

```

1 /*
2  * Does some simple matrix calculations.
3  */
4
5 int8 Array1[3] = {1,2,3}, Array2[3] = {4,5,6};
6 int8 Vector[3][1] = {{7},{7},{7}};
7 int8 Matrix[3][3] = {{1,1,1},{1,1,1},{1,1,1}};
8
9 //returns array[3] = {4,10,18}.
10 Array1 = Array1 * Array2;
11
12 //returns array[3] = {0,1,1};
13 Array1 = Array1 > Array2;
14
15 Matrix[2][0][:2] = Array2[0][:2];
16 Matrix[1][0][:2] = Matrix[0][0][:2] *
17                     Matrix[2][0][:2];
18
19 //returns array[3] = {1,0,0}.
20 Array1 = !Array1;
21
22 //returns array[3] = {-1,0,0}.
23 Array1 = -Array1;
24
25 //Prints strings or variables
26 print "Hello World!";
27 print Matrix[1][1];

```

Source code B.6: Index range operations in CIP.

Formal semantics for CIP

This is the documentation of the formal semantic for scalar and array operations in the CIP programming language, semantic rules not concerned with data parallelism have been omitted, but is reminiscent of the ones described in [32]. For a more thorough explanation of the environment store model used in the rules and the most interesting of the semantic rules see section 4.2.1 and F4-3.

C.1 Formal Semantics

$$[Plus] \quad \frac{env_V, sto \vdash a_1 \rightarrow_a v_1 \quad env_V, sto \vdash a_2 \rightarrow_a v_2}{env_V, sto \vdash a_1 + a_2 \rightarrow_a v} \quad \text{where } v = v_1 + v_2$$

$$\frac{\langle D_A, env_V'', sto[l_1 \mapsto v_1] \cdots [l_n \mapsto v_n] \rangle \rightarrow_{D_A} (env_V', sto')}{\langle y[a] := \{a_1 \cdots a_n\}; D_A, env_V, sto \rangle \rightarrow_{D_A} (env_V', sto')}$$

where $env_V, sto \vdash a_1 \rightarrow_a v_1 \dots a_n \rightarrow_a v_n$
and $l_1 = env_V \text{ next}$
[ArrayDecl] and $l_2 = new \ l_1$
 \vdots
and $l_n = new \ l_n$
and $env_V'' = env_V[y \mapsto l][next \mapsto new \ l_n]$

$$[ArrayLoc] \quad env_V, sto \vdash y \rightarrow_A l \quad \text{where } env_V \ y = l$$

$$[ArrayIndex] \quad \frac{env_V, sto \vdash y \rightarrow_A l}{env_V, sto \vdash y[a] \rightarrow_a v} \quad \text{where } sto \ l + a = v$$

	$env_V, env_P \vdash \langle y := A, sto \rangle \rightarrow_s sto[l_1 \mapsto v_1][l'_1 \mapsto v_2][l''_1 \mapsto v_3][l'''_1 \mapsto v_4]$ <p>where $l_1 = env_V y$ and $sto l_2 = v_1$</p>
$[ArrayArrayAss]$	<p>and $l'_1 = new\ l_1$ and $l'_2 = new\ l_2$ and $sto\ l'_2 = v_2$</p> <p>and $l''_1 = new\ l'_1$ and $l''_2 = new\ l'_2$ and $sto\ l''_2 = v_3$</p> <p>and $l'''_1 = new\ l''_1$ and $l'''_2 = new\ l''_2$ and $sto\ l'''_2 = v_4$</p>
$[ArrayScalarAss]$	$env_V, env_P \vdash \langle y := a, sto \rangle \rightarrow_s sto[l \mapsto v][l' \mapsto v][l'' \mapsto v][l''' \mapsto v]$ <p>where $env_V, sto \vdash a \rightarrow_a v$ and $env_V y = l$</p> <p>and $l' = new\ l$ and $l'' = new\ l'$ and $l''' = new\ l''$</p> $env_V, env_P \vdash \langle y[a_1] := a_2, sto \rangle \rightarrow_s sto[l' \mapsto v]$
$[ArrayIndexAss]$	<p>where $env_V, sto \vdash a_2 \rightarrow_a v$</p> <p>and $l = env_V y$ and $l' = l + a_1$</p> $\frac{env_V, sto \vdash A \rightarrow_A l \quad env_V, sto \vdash a \rightarrow_a v_2}{env_V, sto \vdash A + a \rightarrow_E V}$ <p>where $v_1 = sto\ l$ and $v[0] = v_1 + v_2$</p>
$[ArrayScalarAdd]$	<p>and $l' = new\ l$ and $v'_1 = sto\ l'$ and $v[1] = v'_1 + v_2$</p> <p>and $l'' = new\ l'$ and $v''_1 = sto\ l''$ and $v[2] = v''_1 + v_2$</p> <p>and $l''' = new\ l''$ and $v'''_1 = sto\ l'''$ and $v[2] = v'''_1 + v_2$</p> $\frac{env_V, sto \vdash A_1 \rightarrow_A l_1 \quad env_V, sto \vdash A_2 \rightarrow_A l_2}{env_V, sto \vdash A_1 + A_2 \rightarrow V}$ <p>where $v_1 = sto\ l_1$ and $v_2 = sto\ l_2$ and $V[0] = v_1 + v_2$</p> <p>and $l'_1 = new\ l_1$ and $l'_2 = new\ l_2$</p>
$[ArrayArrayAdd]$	<p>and $v'_1 = sto\ l'_1$ and $v'_2 = sto\ l'_2$ and $V[1] = v'_1 + v'_2$</p> <p>and $l''_1 = new\ l'_1$ and $l''_2 = new\ l'_2$</p> <p>and $v''_1 = sto\ l''_1$ and $v''_2 = sto\ l''_2$ and $V[2] = v''_1 + v''_2$</p> <p>and $l'''_1 = new\ l''_1$ and $l'''_2 = new\ l''_2$</p> <p>and $v'''_1 = sto\ l'''_1$ and $v'''_2 = sto\ l'''_2$ and $V[3] = v'''_1 + v'''_2$</p>

$$\frac{env'_V[x \mapsto l][next \mapsto new\ l], env'_P[p \mapsto (S, x, env'_V, env'_P)] \vdash \langle S, sto[l \mapsto v] \rangle \rightarrow sto'}{env_V, env_P \vdash \langle p(a), sto \rangle \rightarrow sto'}$$

[*ProcCallValue*] where $env_{PP} = (S, x, env'_V, env'_P)$
 and $env_V, sto \vdash a \rightarrow_a v$
 and $l = env_V\ next$

$$\frac{env'_V[y_1 \mapsto l][next \mapsto l'], env''_P \vdash \langle S, sto \rangle \rightarrow sto'}{env_V, env_P \vdash \langle p(y_2), sto \rangle \rightarrow sto'}$$

[*ProcCallRef*] where $env_P\ p = (S, y_1, env'_V, env'_P), env_V\ y = l$
 and $l' = env_V\ next$
 and $env''_P = env_P[p \mapsto (S, y_1, env'_V, env'_P)]$

$$\frac{env_v, sto \vdash y \rightarrow_A l}{env_v, sto \vdash y[a_1] :: [a_2] \rightarrow_A l'}$$

[*RangeOp*] where $l' = l + a_1$

Bibliography

- [1] Saleh Omran M. Hassaballah and Youssef B. Mahdy. A review of simd multimedia extensions and their usage in scientific and engineering applications. *The Computer Journal*, 51(6), 2008. <http://comjnl.oxfordjournals.org/content/51/6/630.full.pdf>. 9, 11, 12
- [2] Shameem Akhter and Jason Roberts. *Multi-Core Programming: Increasing Performance through Software Multi-threading*. Intel Press, 2006. ISBN 978-0-976-48324-3. 11
- [3] Jon Stokes. Simd architectures. <http://arstechnica.com/old/content/2000/03/simd.ars>, 2000. 12
- [4] Rastislav Bodík. *Compiler construction: 14th international conference, CC 2005, held as part of the Joint European Conferences on Theory and Practice of Software*. Springer, 2005. ISBN 978-3-54-025411-9. 11
- [5] Vishal Choudhary Anteneh A. Abbo, Richard P. Kleihorst and Leo Sevat. Power consumption of performance-scaled simd processors. <http://www.springerlink.com/content/0hwqkmrtkv86nmq1/fulltext.pdf>, 2004. 12
- [6] Christoffer Wright. Simd programming. <http://softpixel.com/~cwright/programming/simd/>, 2010. 12, 14, 15, 16, 17
- [7] Xvid codec overview. <http://www.xvid.org/Project-Info.46.0.html>, 2012. 14
- [8] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, December 2011. <http://download.intel.com/products/processor/manual/253665.pdf>. 16, 17, 42
- [9] Intel. Extending the world's most popular processor architecture. 2006. <http://download.intel.com/technology/architecture/new-instructions-paper.pdf>. 16, 17
- [10] The GNU Project. Using the gnu compiler collection. <http://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/>, 2010. 18, 19, 20
- [11] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization - revisited for short simd architectures. 2008. <http://dl.acm.org/citation.cfm?id=1454119&dl=ACM&coll=DL>. 18
- [12] The GNU Project. Auto-vectorization in gcc. <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>, 2011. 19
- [13] Agner Fog. The microarchitecture of intel, amd and via cpus an optimization guide for assembly programmers and compiler makers.

- Technical report, Copenhagen University College of Engineering, 2012. <http://www.agner.org/optimize/microarchitecture.pdf>. 21
- [14] Rajiv Kapoor. Avoiding the cost of branch misprediction. <http://software.intel.com/en-us/articles/avoiding-the-cost-of-branch-misprediction/>, 2009. 21
- [15] Advanced Micro Devices. Amd64 architecture programmer’s manual volume 1:application programming. Technical report, 2009. 21
- [16] Agner Fog. Optimizing subroutines in assembly language an optimization guide for x86 platforms. Technical report, Copenhagen University College of Engineering, 2012. http://www.agner.org/optimize/optimizing_assembly.pdf. 21, 22
- [17] Randal E. Bryant and David O’Hallaron. *Computer Systems A Programmers Perspective*. Pearson, second edition edition, 2011. ISBN 978-0-13-713336-9. 22
- [18] Calvin Lin and Lawrence Snyder. *Principles of Parallel Programming*. Pearson Education, Inc, 2009. ISBN 978-0-321-54942-6. 22, 23
- [19] The SCandAL project. Nesl: A parallel programming language. <http://www.cs.cmu.edu/~scandal/nesl.html>, 2012. 23
- [20] The ZPL team. Overview. <http://www.cs.washington.edu/research/zpl/overview/overview.html>, 2004. 24
- [21] Lawrence Snyder. A programmers guide to zpl. www.cs.washington.edu/research/zpl/zpl_guide.pdf, 1999. 24
- [22] Community. Mono.simd namespace. <http://docs.go-mono.com/?link=N:Mono.Simd>, 2012. 26
- [23] Jaewook Shin. Simd programming by expansion. 2007. www.mcs.anl.gov/uploads/cels/papers/P1425.pdf. 26
- [24] Sablecc. <http://sablecc.org/>, 2011. 26
- [25] Java compiler compiler. <http://javacc.java.net/>, 2012. 27
- [26] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. The compiler generator coco/r. <http://ssw.jku.at/Coco/>, 2011. 27
- [27] Doug Brown, John Levine, and Tony Mason. *Lex & yacc, Second Edition*. O’Reilly Media, Inc., 1992. ISBN 978-1-56592-000-2. 27
- [28] The Flex Project. flex: The fast lexical analyser. flex.sourceforge.net, 2008. 27
- [29] Michael Petter. Cup user’s manual. www2.cs.tum.edu/projects/cup/manual.html, 2006. 27
- [30] Gerwin Klein. Jflex user’s manual. <http://jflex.de/manual.html>, 2009. 27
- [31] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, ninth edition edition, 2009. ISBN 978-0-13-246558-8. 30, 43, 77

- [32] Hans Hüttel. *Transitions and Trees - An Introduction to Structural Operational Semantics*. Cambridge, 2010. ISBN 978-0-521-14709-5. 34, 35, 43, 85
- [33] Tim Hoolihan. Static vs dynamic scope. <http://hoolihan.net/blog-tim/2009/02/17/static-vs-dynamic-scope/>, 2009. 43
- [34] Tiobe index. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>, 2012. 44, 45, 46
- [35] Michaël Van Canneyt. Free pascal: Reference guide. <http://www.freepascal.org/docs-html/ref/ref.html>, 2011. 45
- [36] Corrado Böhm and Guiseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rule. <http://dl.acm.org/citation.cfm?doid=355592.365646>, 1966. 46
- [37] Miran Lipovača. Learn you a haskell for great good!: A beginner's guide. <http://learnyouahaskell.com/>, 2011. ISBN 978-1-593-27283-8. 46
- [38] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objekt orienteret analyse & design*. Marko, 2001. ISBN 978-87-7751-153-0. 47
- [39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 978-0-201-63361-0. 47, 49
- [40] oodesign. <http://www.oodesign.com/>, 2012. 47, 48, 49, 50
- [41] dofactory. <http://www.dofactory.com/>, 2012. 47, 48, 49
- [42] Charles N. Fisher, Ron K. Cytron, and Richer J. LeBlanc, Jr. *Crafting a Compiler*. Pearson, 2010. ISBN 978-0-13-801785-9. 53, 55, 72