A compiler front-end for the WOOL Parallelization library

GEORGIOS VARISTEAS

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:291
A compiler front-end for the WOOL Parallelization library

Georgios Varisteas
yorgos(@)kth.se

15 October, 2010

A master thesis project conducted at

Examiner:
Mats Brorsson
Supervisor:
Karl-Filip Faxén
Abstract

WOOL is a C parallelization library developed at SICS by Karl-Filip Faxén. It provides the tools for developing fine grained independent task based parallel applications. This library is distinguished from other similar projects by being really fast and light; it manages to spawn and synchronize tasks in under 20 cycles.

However, all software development frameworks which expose radically new functionality to a programming language, gain a lot by having a compiler to encapsulate and implement them. WOOL does not differ from this category. This project is about the development of a source-to-source compiler for the WOOL parallelization library, supporting an extension of the C language with new syntax that implements the WOOL API, transforming it and eventually outputting GNU C code. Additionally, this compiler is augmented with a wrapper script that performs compilation to machine code by using GCC. This script is configurable and fully automatic.

The main advantage gained from this project is to satisfy the need for less overhead in software development with WOOL. The simplified syntax results in faster and more economical code writing while being less error-prone. Moreover, this compiler enables the future addition of many more features not applicable with the current state of WOOL as a library.
Acknowledgements

I would like to give special thanks to...

Christian Schulte, Associate Professor at KTH
for introducing me to real programming

Mats Brorsson, Professor at KTH and my Examiner
for giving me clear perspective

Karl-Filip Faxèn, Researcher at SICS and my Supervisor
for being patient and always accessible

Frej Drejhammar, Researcher at SICS
for sharing his wisdom on Linux and C

Dimitri Mazmanov, Student at KTH
for being a reliable partner in all the courses we had together

SICS,
for providing a welcoming working environment and the big
amounts of coffee required for this project

My mother Eletheria,
for always supporting me on my decisions and allowing me to
accomplish my goals

Georgia,
for being my oasis.

...As well as everyone else who listened to my questions and spent
the time to help me along the way of completing this project
## Contents

1 Introduction .............................................. 1  
  1.1 Problem statement ........................................ 1  
  1.2 Goals .................................................. 2  

2 Background .................................................... 3  
  2.1 General introduction to compilers .......................... 3  
    2.1.1 Scope .................................................. 4  
    2.1.2 A compiler front-end .................................. 4  
    2.1.3 A compiler back-end .................................. 6  
    2.1.4 Preprocessing .......................................... 7  
  2.2 Related work .............................................. 8  
    2.2.1 LLVM and CLANG ....................................... 8  
    2.2.2 Rose .................................................. 8  
    2.2.3 Cetus ................................................ 9  

3 WOOL’s original API and WOOL C .......................... 10  
  3.1 Types and the original WOOL syntax ........................ 10  
  3.2 A TASK definition .......................................... 11  
  3.3 The SPAWN operation ....................................... 12  
  3.4 The SYNC operation ........................................ 12  
  3.5 The CALL operation ........................................ 13  
  3.6 A parallel FOR loop definition ........................... 13  

4 Front-end: A GNU C compatible parser ...................... 17  
  4.1 Lexical analysis .......................................... 17  
    4.1.1 Scanning GNU extensions .............................. 20  
    4.1.2 Scanning the WOOL language .......................... 21  
  4.2 Syntactic analysis ........................................ 21  
    4.2.1 The dangling else ...................................... 22  
    4.2.2 C preprocessor directives ............................ 23  
    4.2.3 Inline assembly ........................................ 23  
    4.2.4 __extension__ .......................................... 24  
    4.2.5 GNU Attributes ........................................ 24  
    4.2.6Unnamed fields ......................................... 25  
    4.2.7 Alternative keywords .................................. 25  
  4.3 The parser’s symbol table ................................. 25
4.4 Identifying and handling scope ........................................ 29
  4.4.1 Built-in functions & types ....................................... 30
4.5 Intermediate representation ........................................... 31

5 Back-end: Analysis and transformations .............................. 35
  5.1 Code transformations ................................................ 36
  5.2 Semantic Analysis revisited ........................................ 36
  5.3 Target language printing ............................................ 38
  5.4 The declarations table .............................................. 38

6 Evaluation ......................................................................... 40
  6.0.1 Generated code quality ............................................ 40
  6.0.2 Performance issues ................................................ 48

7 Conclusions .................................................................. 51
  7.1 Conclusion ............................................................... 51
  7.2 Future work ............................................................. 51
    7.2.1 GNU extensions .................................................. 52
    7.2.2 C macro Include directives .................................... 52
    7.2.3 Type analysis ..................................................... 53
    7.2.4 Optimize specific patterns ...................................... 53
    7.2.5 Output C code rather than C macros ....................... 53
    7.2.6 A generic sync ................................................ 54
    7.2.7 The prototype issue ........................................... 54

A The final vocabulary ..................................................... 56

B The final grammar ....................................................... 62

C The Abstract Syntax Tree ............................................... 79

D The TAB-table structure ................................................. 83

E The stack structure ....................................................... 85
  E.1 An example: ScopeStack ........................................ 85

F WCC: The wrapper script ............................................... 86
### List of Figures

1.1 The multiple steps needed to compile a **WOOL** project. ................................. 2

2.1 A vague decomposition of a compiler. ................................................................. 3

2.2 The three-part process of a front-end. ............................................................... 4

2.3 The three-part process of a back-end. ............................................................... 6

3.1 The **TASK** definition’s original syntax. ........................................................ 11

3.2 A **TASK** definition’s new syntax ................................................................. 11

3.3 Examples of task definitions. ................................................................. 11

3.4 The **SPAWN** operation’s original syntax. .................................................... 12

3.5 The **SPAWN** operation’s new syntax ............................................................ 12

3.6 Examples of usages of the **SPAWN** operation. ........................................... 12

3.7 The **SYNC** operation’s original syntax. ...................................................... 12

3.8 The **SYNC** operation’s new syntax ............................................................. 12

3.9 Examples of usages of the **SYNC** operation. ............................................. 13

3.10 The **CALL** operation’s original syntax. ...................................................... 13

3.11 The **CALL** operation’s new syntax ............................................................. 13

3.12 Examples of usages of the **CALL** operation. ............................................ 13

3.13 The **parallel FOR loop**’s original syntax. .................................................. 14

3.14 The **parallel-FOR-loop** definition’s new syntax. ......................................... 15

3.15 A **parallel FOR loop** initialization definition. ........................................... 15

3.16 A **parallel FOR loop** test definition. .......................................................... 15

3.17 A **parallel FOR loop** step definition. .......................................................... 16

4.1 A sample of shadowing that creates scanning conflicts ................................. 18

4.2 Scanning GNU inline assembly instructions. ................................................... 21

4.3 Scanning GNU attributes. ................................................................. 21

4.4 Scanning the **WOOL** language. ................................................................. 21

4.5 The dangling else conflict ................................................................. 22

4.6 Solving the dangling else conflict ............................................................... 22

4.7 Syntactically supporting the **inline assembler** instructions ....................... 23

4.8 Syntactically using the **inline assembler** instructions .............................. 23

4.9 Syntactically supporting the **_extension_** GNU keyword. ............................ 24

4.10 Syntactically supporting GNU **attributes**. .................................................. 24

4.11 Syntactically supporting the **unnamed fields** GNU extension. ..................... 25

4.12 Syntactically supporting the GNU **alternative keywords**. .......................... 25

4.13 A graphical representation of a sample **symbol-table**. .............................. 27
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.14</td>
<td>Storing a symbol in the <em>symbol-table</em>.</td>
</tr>
<tr>
<td>4.15</td>
<td>Keeping track of parameter lists for the <em>symbol-table</em>.</td>
</tr>
<tr>
<td>4.16</td>
<td>Terminating the scope of a function prototype’s parameter-list.</td>
</tr>
<tr>
<td>4.17</td>
<td>The fundamental structures of the AST.</td>
</tr>
<tr>
<td>4.18</td>
<td>An abstract sample of the AST backbone.</td>
</tr>
<tr>
<td>4.19</td>
<td><em>translation-unit</em> grammatical rule.</td>
</tr>
<tr>
<td>4.20</td>
<td>An abstract sample of a tree which includes a list.</td>
</tr>
<tr>
<td>5.1</td>
<td>The three-part process of a back-end.</td>
</tr>
<tr>
<td>5.2</td>
<td>Examples of nested and multi-symbol declarations.</td>
</tr>
<tr>
<td>5.3</td>
<td>Samples of printing fallacies and pitfalls.</td>
</tr>
<tr>
<td>6.1</td>
<td>Formal specification of the code translation process, part I</td>
</tr>
<tr>
<td>6.2</td>
<td>Formal specification of the code translation process, part II</td>
</tr>
<tr>
<td>6.3</td>
<td>The <em>fib</em> program written in WOOL C</td>
</tr>
<tr>
<td>6.4</td>
<td>The <em>fib</em> program translated into GNU C.</td>
</tr>
<tr>
<td>6.5</td>
<td>The <em>loop</em> program written in WOOL C</td>
</tr>
<tr>
<td>6.6</td>
<td>The <em>loop</em> program translated into GNU C.</td>
</tr>
<tr>
<td>6.7</td>
<td>The <em>knapsack</em> program written in WOOL C, part I</td>
</tr>
<tr>
<td>6.8</td>
<td>The <em>knapsack</em> program written in WOOL C, part II</td>
</tr>
<tr>
<td>6.9</td>
<td>The <em>knapsack</em> program translated into GNU C, part I.</td>
</tr>
<tr>
<td>6.10</td>
<td>The <em>knapsack</em> program translated into GNU C, part II.</td>
</tr>
<tr>
<td>6.11</td>
<td>Plot: Execution time over lines of code.</td>
</tr>
<tr>
<td>6.12</td>
<td>Plot: Execution time over the total number of tasks.</td>
</tr>
<tr>
<td>6.13</td>
<td>Plot: Execution time over the maximum arity.</td>
</tr>
<tr>
<td>7.1</td>
<td>A new WOOL task syntax to solve the prototype issue.</td>
</tr>
<tr>
<td>7.2</td>
<td>A sample task prototype translation.</td>
</tr>
<tr>
<td>D.1</td>
<td>Graphical representation of a sample <em>TAB-table</em>.</td>
</tr>
<tr>
<td>F.1</td>
<td>WCC usage information.</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Non-terminals whose productions perform scope initialization and termination. .................. 29
4.2 Dummy non-terminals used to initialize and terminate scopes. ................................. 30

6.1 Execution time comparison between CilkC, WCC and WPP. All times are given in milliseconds.
   Each result is the mean average of 10 runs. ................................................................. 48
6.2 Details on the programs used for comparing CilkC and WCC. Lines of code counted with David
   A. Wheeler’s SLOCCount, version 2.26. ........................................................................ 49
1 Introduction

1.1 Problem statement

WOOL is a C library, created by Karl-Filip Faxén at SICS [9], which enables an application to take advantage of multiple cores while achieving noteworthy speedup. WOOL manages this by handling thread creation and scheduling among the available cores. It defines specific programming constructs that a developer can use; namely tasks, as functions to be executed asynchronously, and automatically parallelizable loops. Currently, WOOL consists of the library source file, a static header file and a bash script which generates a dynamic header-file. The static header file includes all the declarations that are needed by the library itself as well as all external library dependencies. The dynamic header file corresponds to the code using the library, thus it is dependent on it’s specifics and has to be generated for each project. In that respect, a script that dynamically generates this header file based on passed arguments is a viable choice considering all alternatives, especially comparing to having the library bloated with unnecessary code that tries to predict all possible uses, which will always prove to be inadequate at the end; however, it does not negate the uncommon extra overhead.

Nevertheless, the list of problems with WOOL being a standalone library, is not confined to just having to generate a new header file very often. In order to use the library the developer has to follow a weird and cumbersome syntax that also prohibits the use of a custom main function. Specifically tasks and loops, require the hard coding of several parameters which far exceed the normal requirements of such constructs; these include the specification of all the free variables in a parallel loop body or passing the loop iteration boundaries as arguments instead of a conditional expression. This poor syntax makes WOOL development slow and unintuitive.

But there are other secondary problems too. A standalone library can do little to provide semantic error checking and reporting. This is a major drawback that prohibits the adoption of WOOL in the development of complex parallel systems.

Are all these problems unavoidable? Actually, it is common practice for a compiler to immediately solve most of these problems or at least it provide the foundation for the appropriate features to be added. So, the main purpose of this project is to create exactly that, a compiler that understands WOOL code. In this perspective, this thesis project is about designing a user-friendly, dedicated syntax that implements the WOOL API as a new language, called WOOL C, and developing a compiler to support it, which will also provide mechanisms for automatically generating the dynamic header file and checking for WOOL related semantic errors. However, it was deemed unnecessary to proceed with a full compiler that generates machine code. On the contrary it would be sufficient to transform the new syntax to standard C code; the same C code that the developer would have to write if a compiler was not used. This compiler should accept as input the new language and be able to translate it into C.

To sum everything up, in this project I have tried to resolve the afore mentioned problems by developing a dedicated source-to-source compiler, called WPP, accepting WOOL C as input and producing standard GNU C as output. The new language is going to be an extension to the syntax of GNU C. The reasons for selecting GNU C as the target language are mainly portability and scalability of the compiler but these are explained in more detail later on.

Using the WOOL library in its original version, the developer needs only to generate the dynamic header file,
include both header files and then compile the source code, linking also with the library object file. This process becomes a bit more complicated with the introduction of the new compiler. The individual steps are shown in the figure that follows. For reasons explained later, the new compiler expects to receive a preprocessed input. So, the static header file and all other external library dependencies have to be resolved in advance by the C preprocessor. Then it is analysed and transformed to GNU C. Next, it is merged with the dynamic header file, which will automatically be generated according to the analysis results. The final step is compilation to machine code also linking everything with the pre-compiled WOOL library.

The introduction of a dedicated language with its compiler, although complicating the compilation process, is an improvement for many reasons. First, a script, called WCC, is provided which can automatically execute all steps. Second, coding in WOOL C will significantly reduce the overhead for the developer, in time and resources. All in all, from the user’s perspective the whole process becomes much friendlier.

![Figure 1.1: The multiple steps needed to compile a WOOL project.](image)

1. Design a new language extension to GNU C, which implements all the functionality of the WOOL library. The complete new language is going to be referred to as WOOL C.

2. Develop a parser that supports as much of the GNU C language as possible. This parser should handle lexical and syntactic analysis.

3. Extend the parser to support the WOOL C language extensions.

4. Semantically analyse the given input code as to extract the WOOL related parts of it. Determine all necessary information in order to transform those pieces of code to standard GNU C.

5. Develop a wrapper script, WCC, to encapsulate the whole process and provide a single step full compilation of the given code, by using a pre-installed C compiler. The script is pre-configured to use GCC; however, all settings can be adapted to support any C compiler.
2 Background

2.1 General introduction to compilers

A compiler is a program that takes as input the specification of a program and produces the specification for another, equivalent program [6]. A compiler is composed out of two main parts; the front-end that analyses the source language and the back-end that maps it to the target language as depicted in figure 2.1. Any source language has to maintain a conceptual clarity in its syntax and general phrasing. This is accomplished mainly with the use of plenty of punctuation symbols, but also with the help of very elaborate although strict syntactic rules. However, once the concepts described by a source program have been understood by the compiler, all of the punctuation and other rules become totally unnecessary for the continuation of the process.

There is a common abuse of the term compiler to refer to a suite of tools that are used in conjunction to the compiler in order to generate an executable application; this misconception has been generated because their use is usually transparent to their user; this is either because all these tools are incorporated into one application or because the compiler invokes them in the background as needed. In reality the compiler itself is limited to the single task of translating from one language to another.

The front-end’s role is to understand the source program and strip it of all punctuation while transforming its language into a more generic and abstract notation. Such notation is called an intermediate representation (IR) and it is a tree language [5], as its sentences form an abstract tree structure denoting strict parent-child relationships. The back-end receives the IR as input and is able to analyse, optimize and transform it into any other language. Most often the target language is assembly.

Nevertheless, for the purposes of this project assembly code generation is not required. Instead, the target language is just another high level source language. Such a compiler is called a source-to-source compiler and it is usually regarded as front-end to the compiler which eventually translates the program into assembly code. The general reasons for creating such a limited compiler are plenty; amongst them the most important, which also apply to this case, are simply convenience and development time constraints.

To elaborate more, first of all it would be really inefficient to produce a full compiler as it would be impossible to implement the rich amount of optimizations that other popular compilers do today, thus losing a lot of performance for the produced executables. Secondly, the process of mapping one language to another is not a trivial process and its requirements in development time and resources are analogous on the foreignness of the two languages; thus translating a high level language to machine code is really demanding. Moreover, since the input language would be nothing more than a small extension to an already heavily supported language, GNU C, translating is rather straightforward.

However, this task is not as simple as it may seem. A source-to-source compiler in itself is divided into the
same components. A front-end to handle the source language and a back-end to map it to the target language. In between there is also one more intermediate representation. Since every programming language has a lot of syntactic quirks, the front-end extracts only the necessary parts as an abstract and generalized tree structure. This structure is called an Abstract Syntax Tree and is the first step to creating the intermediate representation. The intermediate representation is the most abstract model of the source program possible, thus allowing to map its conceptual constructs to a totally new language. In many cases though, where the two languages are not that different, there is no need to really abstract this model; so, usually the intermediate representation is the Abstract Syntax Tree itself, or something really close to it. In all cases, such a generic tree structure can easily be translated into another programming language, the target language.

2.1.1 Scope

Before moving forward, there is one concept that needs special mention in regard to how it is used and measured in this project. This is the concept of scope. Scope denotes the range of effect of a declared entity. All programming entities, for example a function or a variable, have their specific scope. Outside of that scope the entity has no effect or most commonly it doesn’t even exist. In C a special scope is called global scope and it is the root scope that is outside of any other scope. A C program is defined as inside a unique global scope; this way all other scopes are nested to it eventually creating a tree structure of nested scopes.

In this project scopes are labelled by integer numbers. The global scope is assigned the number 0 and all nested scopes are defined by their absolute distance from it. From here on, the lowest scope refers to the global scope and the highest scope to the innermost scope; also, when a scope is described as lower that another it means that it is a parent scope.

A final note about scopes is to answer how ambiguousness is avoided since two nested scopes of the same distance to the root are going to be defined with the same integer value. The reasoning is simple and it is heavily vested on how both C and a compiler works; C is an imperative language and as such it follows a strict execution path which is inherently sequential. This means that at any given time only one scope is active. The compiler follows the same sequential nature. This means that no scope is to become the subject of any component of the compiler until processing of any other has finished (for higher scopes) or suspended (for lower scopes).

2.1.2 A compiler front-end

![Figure 2.2: The three-part process of a front-end.](image)

At first thought it might seem simple to extract the necessary information from a piece of source code. However, when one tries to think about the worst possible complexity that this text might have, then the process starts to seem even impossible. This is why reading the source code is not a plain and straightforward algorithm. A compiler’s front-end is composed out of three modules, performing three discrete analysis steps.

The first module is the scanner that performs lexical analysis on the source program to produce a stream of tokens that represents it. This is called tokenization. In order to do that a vocabulary of the language is needed.
The vocabulary is a specification of the words that exist in the language. This should be recognized and understood by the scanner. Inside the scanner, each word is statically assigned to a token. There are occasions where the same token is shared among multiple words. This is because in some languages, including C, multiple keywords express the exact same semantic concept, playing the role of alternative to suit different development habits and notational standards. The scanner uses regular expression patterns to recognize and match these words. It reads from the input file one character at a time, aggregating them until a pattern is matched. Then its respective token is returned and the scanner continues the same way until the end of the input.

The stream of tokens that the scanner produces is then syntactically analysed by the parser. This analysis is based upon a grammar of the implemented language. This grammar consists of rules that define all possible valid sentences of the language. Each sentence is made up from a combination of tokens and it defines a new token which can be used for the definition of another sentence. This recursive hierarchy starts with a root token (usually called the translation_unit) that represents the whole input program, expanding to a tree structure that is comprised by two sets of nodes. At the bottom there are the leaf nodes, called terminals, each corresponding to a token. Between the root and the leaf nodes there are the non-terminals that represent conceptual aggregations of tokens and other non-terminals thus forming a valid sentence.

There are two different ways of performing semantic analysis, Top-Down and Bottom-Up. The first way is about starting from the root node and then extending the tree downward by matching the tokens as accepted from the scanner to the first appropriate rule in the grammar. It accepts tokens hoping to match the selected rule and continue by applying more rules. If at any point a token is found that breaks the rule at hand, the parser backtrack to see if selecting a different rule at an earlier point would make a difference. By making a grammar non ambiguous the need for backtracking can be eliminated thus streamlining the whole process.

Bottom-Up parsers work in the exact opposite way, by assigning tokens as leaves to the tree and then trying to match them to rules, which become their parents, eventually reaching the root node. In other words at any given point the parser has buffered a group of consecutive tokens and it tries to match them to a sentence of the grammar, eventually replacing them with the single token of that sentence; this is called a reduce operation. If there are multiple sentences matching the sequence the parser asks for the next token, this is called a shift operation. Finally if no matching sentence is found then the parser terminates with a syntactic error determining that the given sequence is not correct in this language.

In this project I have implemented a Bottom-Up parser. These parsers can be implemented in multiple ways mainly according to the type of grammar they are based on. There are two general types of grammars for such parsers. LR(k) and precedence parsers, with the first presenting a number of significant advantages of the latter [12] [25]. The name LR is derived from Left to right, Rightmost derivation with k look-ahead symbols which in a nutshell describes how these parsers work. However, this project used a subclass of the LR family, specifically a LALR(1) parser. These parsers constitute a refinement over the typical LR parser although they cannot handle the same levels of language complexity. However, they are sufficient for the C programming language [5]. LALR was selected because it provides the following advantages over other types:

1. The parser application can be automatically generated from an LALR grammar.
2. An LALR parser is always more compact.
3. An LALR parser is linear in speed to the size of the input.
4. An LALR grammar can be used as documentation of the language being recognized.

A special note on the first item on the list is required as it allows for automated parser generators, usually
called compiler-compilers. One of these generators is GNU Bison which is compatible with the proven YACC [13], offering some nice new features. Bison was used in this project to generate the parser. It defines its own notation for implementing language grammars that resembles a lot the formal way of presenting a grammar. It generates feature rich, fast and compact parsers, thus negating any advantage from manual development [10].

Moving on, it was implied that the parser creates a tree structure that represents the given source file. This tree aims to minimize most of the excess information in the input source and also to eliminate any semantic ambiguity. This tree is called the abstract-syntax-tree or AST. The third step in the front-end, semantic analysis, tries to minimize the information stored in this tree even further, eventually producing a new tree construct. This is called an intermediate representation (IR) and it often does not resemble the input language at all. Each node in this tree has been reduced to represent just a computable operation. The collection of possible such operations are closer to the machine and might be no more complicated than numerical addition and subtractions, plus memory load and store functions. This IR is then passed on to the back-end.

### 2.1.3 A compiler back-end

![Diagram](https://example.com Diagram.png)

**Figure 2.3:** The three-part process of a back-end.

The back-end is responsible for code generation. In other words it analyses the given program and emits new code in a different language. When talking about compilers in general this is usually assembly language, thus allowing to create an executable after assembling and linking; but as with this project, the target language, can actually be any language. The back-end is also comprised of three distinct steps. First comes instruction selection which selects the appropriate instructions of the target language to replace the operations described by the IR nodes. The second step is scheduling which decides on the order of execution of the target language instructions. Finally, comes register allocation which appoints a register for each value taking part in the instructions.

However, when talking about a source-to-source compiler things are dramatically different. Transforming a source language to machine language, requires transferring the defined concepts of the source language through multiple levels of increased complexity. This translates to a simple operation in the source language corresponding to hundreds of machine code instructions, making it a very complicated process to be performed efficiently, in the end. Two high level source languages on the other hand, share more or less the same amount of complexity for their operations, making the mapping close to 1-1. In other words, source-to-source compiling typically means replacing one operation with just one or two operations of the same complexity overall.

This makes the overall process of the back-end quite simpler and vastly less demanding. First of all, a source-to-source compiler’s back-end needs not do register allocation. Then instruction selection is done almost in combination to scheduling as there is not much need for advanced ordering optimizations. This will be done anyway by the final compiler in its own scheduling process.

The final part done in the back-end is to print the new language taking care of reintroducing punctuation. This is not trivial at all as a single wrongly placed white space character can render the whole program false or even

---

1. It is also common for scheduling to be repeated after register allocation or even omitted the first time. For a complete reference consult [4]
General introduction to compilers

worse, syntactically correct but with errors.

2.1.4 Preprocessing

An important part of programming in the C language is the use of preprocessor directives. These are included in a C source file, commonly mixed with C code although they are not actually part of the C language. Most directives have the power to drastically alter a program. The most used directives are macros, which replace one piece of text with another. Although simple, they have remarkable power. Their typical use includes extracting pieces of code that are repeated through a project to a single macro; that way the actual code is written only once in the macro definition while at its place there is a simple unique placeholder. Applying such directives is called expanding a macro exactly because the code replacing the macro is almost always significantly bigger than its placeholder. Preprocessing is not restricted to replacing text and provides many other directives with richer functionality; however it is not the purpose of this text to describe them further, so for more information the reader could consult any thorough manual of the C language.

On the other hand there is one preprocessor directive which is of great importance to this project. This is include. This directive merges a source file with others. Most often the files to be merged are headers of external libraries. Removing this functionality would make it a real hardship to use external libraries in any C project. Since C is not a very high level language, even the most basic programming functionality is not part of the language but has to be developed; this is where the various external libraries come in, making their role an absolute necessity for large projects.

The preprocessor directives are a tricky part. In order to parse and implement them the compiler would require to go on extreme lengths of understanding and manipulating C code. Since there are plenty of C preprocessors it would be useless to try and duplicate there behaviour so it was decided from the start to exclude support for preprocessing directives from this project. In other words they are simply discarded by the scanner. However, there is one problem with this. Since it is these directives that piece together a C program split among many files or its dependencies over external libraries, include directives cannot simply be ignored. Not any other code altering directives for that matter. So it is mandatory to have all source code be preprocessed before parsing begins. This means that all external libraries will also be merged into the file before parsing.

A big part of the C language are declarations. A very important rule governing declarations is that they cannot be defined twice in the same scope. This means that it is of great importance not to merge a source file with any other more than once. Also, since external library dependencies are commonly shared among other libraries, the fear of including a file multiple times is real and very frequent. The preprocessor defines mechanisms to avoid such situations. However, once two files are merged, any safety mechanism has been discarded, meaning that if such operations are performed asynchronously it is very possible for shared dependencies to be merged again.

To avoid such scenarios it is imperative for all dependencies to be handled at the same time. To accommodate this behaviour the WOOL library defines two header files. A static one which includes all declarations needed by the library plus all external dependencies. This file can be included before any other processing takes place, eventually allowing the preprocessor to have the complete list of dependencies at its disposal. The second header file, the dynamic header file, includes no such dependencies, hence it can be merged and processed at any later stage.
The idea of a source to source compilers is not truly novel. The reasons for writing such a compiler are plenty, language extensions being a common one. Optimization, portability and maintainability being popular others. Many frameworks exist allowing easy construction of custom such compilers, each with its own approach to the subject. The common denominator is to handle almost all significant analysis of pre-supported source languages, allowing their user to focus on adding only the specific new features desired.

However, using a ready-made framework for building any project, translates into two major characteristics; first the end-project’s features are constrained to what the framework has to offer; second, it is certain that the end-project will inherit all traits of the framework’s components used, good or bad. On average there is no one solution to fit all, on the contrary most are quite inefficient; it comes down to being very subjective, weighting out the benefits of each one and choosing the less damaging.

The most significant frameworks for building custom compilers are presented next.

2.2.1 LLVM and CLANG

The Low Level Virtual Machine (LLVM) is a compiler infrastructure, written in C++, which offers several levels of optimizations for programs written in several languages [17]. It was originally designed as more aggressive, higher performance system for the GCC toolkit. As such it can accept the GCC’s intermediate form (IF) and optimize it, eventually producing either an optimized IF or machine code.

Although, its original purpose was the compilation of the C and C++ languages, its open design allowed it to acquire, in time, a rich collection of front-ends supporting plenty of languages [18]. The most nominal among them are Objective-C, Fortran, Ada, Haskell, Java bytecode, Python, Ruby and ActionScript.

One of the most important parts of LLVM is the CLANG front-end, providing extensive support for C, C++, Objective C and Objective C++ languages [1]. Internally, CLANG follows a library-based architecture and has been design so that these libraries can be combined in many different ways to provide different tools. One of the possibilities is to give a source-to-source compiler between any of the supported languages.

However, one feature not supported is the ability to teach CLANG new languages. Although, having a C parser and AST ready-made is a great advantage, hacking CLANG internals manually would be a very challenging task with minimal gain in respect to generally applicable experience and knowledge. Furthermore, basing this project on CLANG would restrict it to the platforms where LLVM is available. It is unfortunate that LLVM does not yet enjoy a widespread usage in the Unix community making the aforementioned restriction quite important.

In overall, LLVM is a great platform for building compiler front-ends, especially considering the extensive documentation available. It is faster and memory-friendlier than GCC. In regard to this thesis project though, portability was deemed a very important factor, thus balancing the scale towards the generic and independent compiler which was eventually developed.

2.2.2 Rose

Rose is a dedicated framework for building source-to-source translators [22]. It provides ready made libraries for a wide variety of analysis and optimization tasks. Being dedicated in producing source-to-source translators,
the ROSE framework’s internals are specially designed for this purpose leaving very little to be desired. For language support it uses a collection of several independent front-ends and back-ends all supporting the same IR, which can be combined in any way to produce arbitrary source-to-source compilers.

Specifically for the C language family it uses the Edison Design Group’s C++ front-end (EDG) [2]; this was designed mainly to support the C++ language of the ISO/IEC 14882:2003 standard, although it includes support for ANSI/ISO C (both C89 and C99, and the Embedded C TR), the Microsoft dialects of C and C++, GNU C and C++, Sun C++, the cfront 2.1 and 3.0.n dialects of C++, and K&R/pcc C.

Moreover, ROSE is built around the SAGE III intermediate representation which is very successful in modelling C++, thus also C, source programs. It provides full support for C preprocessor control structures and even comments which can be appropriately transferred through the compiler to the output source language [21].

A selling point for ROSE is the initial design principle to allow non-experts to built their own custom compilers. However, like with all other related projects, the actual purpose is to allow custom processing and optimization for a supported language, rather than the development of new languages. In other words, using ROSE for this project would mean either the development of a new front-end or the modification of EDG. Given the complexity of SAGE III IR and considering that EDG is around 500,000 lines of code, it is clear to assume that once more such a solution would prove very demanding but also inefficient for a non-professional, one-man project.

### 2.2.3 Cetus

The final member in the list is Cetus. It is a source-to-source restructuring compiler infrastructure for C programs, and is a follow-on project to the Polaris Fortran translator [7]. Initially built by students of Purdue university, its main focus has been from the start to provide advanced optimizations for parallel C programs, mainly around the OpenMP dialect.

Cetus’ C parser is based on antlr and as such it is written in Java. Both these characteristics make it easy to extend and modify its support to the C language. The downside though is that it supports only ANSI C, leaving much to be desired before ISO C, then GNU C and eventually WOOL C support could be incorporated. In contrast, this project set from the start the base language to be GNU C, the most commonly used superset of the C language.

Emphasis is given on the intermediate representation and type system in Cetus [14]. The feature rich, expressive IR is also easy to maintain, easily understood by non-experts and most importantly extendible and modifiable. The type system is clear and flexible allowing to easily map it to the highly complex type system of languages like C.
WOOL’s original API and WOOL C

The API implemented by WOOL is composed of five constructs for defining, spawning and syncing tasks. A task corresponds to a thread in the work-stealing terminology, and is a sequential program of its own right that is executed in multi-threaded shared memory model [11]. These constructs are:

1. A TASK definition.
2. The SPAWN operation.
3. The SYNC operation.
4. The CALL operation.
5. A parallel FOR loop definition.

As a library WOOL makes heavy use of the C preprocessor macros. Specifically, its whole API is nothing more than applications of such macros. What this means is that there is no real intelligence in the transformations of the macros to actual C code by the C preprocessor. For that reason, most of these macros require parameters that are far from the typical information that a developer keeps track of when using similar frameworks and libraries.

For each of the above constructs, I have devised new syntax with primary focus on simplification and familiarity to mainstream paradigms. These goals were achieved by attacking the problem in two ways. First, by exploiting the presence of a compiler, which can automatically figure out most of the information that the original syntax requires the developer to calculate. This way the new syntax is stripped only to the bare minimum. Second, by removing all relation to macros and designing the new syntax as it is part of the C language itself, making use of existing constructs as much as possible.

Each of the sections that follow, presents one of the aforementioned constructs. In each of them, first the original syntax is shown and explained, immediately followed by the new syntax designed in this project. I selected the approach of combining the two in order to enable easy comparison between them and to provide a single, unified, thorough documentation on the usage of WOOL through this compiler. The fact that WPP only converts the new syntax to the old, makes the comparison necessary for every effort to extend the compiler with new features or simply keep up with the independent development of WOOL itself.

3.1 Types and the original WOOL syntax

A special note has to be given on what I consider one of the most important quirks of the original syntax of WOOL. That is its definition of type. In C a declaration is composed of the type and the declarator. The declarator in turn comprises of the pointer and the direct declarator, with the latter having its own big hierarchy of components, one of which is the declared symbol (usually called the identifier). This complicated grammatical structure is common amongst most languages in the C family.

What is important regarding WOOL though, is the fact that when referring to a name as part of a declaration of one of its constructs, it means only the declared symbol of the underlying C declaration. The type, the pointer and most of the grammatical entities following the symbol are all together referred to as the type.


### 3.2 A TASK definition

Conceptually a TASK definition is similar to that of a function. There is a declaration and a body; the declaration includes a unique identifier and a parameter list. Syntactically though, TASK definitions are originally quite different to common functions and this is mostly because of the nature of macros. Macro usages act much like function calls; so any information specific to such a call can be nothing else than a parameter to it. Specifically for tasks this means that the return type, the identifier and the parameters are all part of the same parameter list.

```
1 TASK_n ( type, name[,type,name[,...]] ) { â€œ }
2 VOID_TASK_n ( name[,type,name[,...]] ) { â€œ }
```

**Figure 3.1:** The TASK definition’s original syntax. The second case is used when the return type is void. The optional part corresponds to a comma-delimited list of all the arguments to the task; these are given as <type,name> tuples. The <n> corresponds to the number of arguments.

The original definition of a task is defined using the macro TASK_<n>, where <n> is a place-holder explained later. This macro accepts a number of parameters; first is the task’s return type, next is its unique identifier and then, optionally, a list of arguments; these arguments are expected in the form of comma separated tuples of <type, name>. The place-holder <n> is the number of arguments and it will be referred to as the *arity* of a task.

Finally, in the special case that a task returns void, a special macro needs to be used, VOID_TASK_<n>. This follows the same syntax as TASK_<n> with the exception of omitting the first argument of the return type.

Having the arity be part of the name of the macro call means that there is a dedicated macro for each one. This the most important reason for the existence of the dynamic header file, as it contains exactly these macro declarations according to the specific needs of each source file.

As mentioned in the beginning, a task is nothing more than a specialized function. This is why for the purposes of the new syntax there is no need for any elaborate syntax other than a simple keyword to signify the special nature of the function. Such keyword prefixes every task definition. Other than that, the new syntax is identical to that of a normal C function.

```
1 wool_task_definition
2     : 'wool' function_definition
3         ;
```

**Figure 3.2:** A TASK definition’s new syntax.

The figure below shows examples of task definitions using both the original and new syntax. In each example the original syntax is

1. TASK_2 ( int, main, int, argc, char **, argv ) { â€œ }
wool int main ( int argc, char **argv ) { â€œ }

2. VOID_TASK_2 ( taskname, int, argl, char *, arg2) { â€œ }
wool void taskname ( int argl, char *arg2 ) { â€œ }

**Figure 3.3:** Examples of task definitions.
### 3.3 The SPAWN operation

`SPAWN` is responsible for instantiating a task. It is much like a specialized function call in that it is asynchronous. Although a task can have a return value, the `SPAWN` operation does not return it. The `SYNC` operation is responsible for that.

As a function call, `SPAWN` requires knowledge of the task identifier to be called and of course any arguments it may require. The original syntax, figure 3.4, is not that much different to the new syntax, figure 3.5. However, it was intended to try and make the syntax resemble a normal function call as much as possible. In both cases the operation expects the name of the task to call and optionally a parameter list.

```
1 SPAWN ( name[,argument1[,...]] );
```

**Figure 3.4:** The `SPAWN` operation’s original syntax.

```
1 wool_spawn_statement
2     : 'wool_spawn' IDENTIFIER '(' expression ')' ;
```

**Figure 3.5:** The `SPAWN` operation’s new syntax

```
1. wool_spawn task1(n-1) ;
SPAWN (task1, n-1) ;
2. wool_spawn task2( ) ;
SPAWN (task2) ;
```

**Figure 3.6:** Examples of usages of the `SPAWN` operation.

### 3.4 The SYNC operation

`SYNC` is responsible for synchronizing the main execution with the most recently spawned task. This means that the order of synchronization is reversed to the order that the tasks were spawned\(^1\). `SYNC` also returns the return value of the task.

```
1 SYNC ( name )
```

**Figure 3.7:** The `SYNC` operation’s original syntax.

```
1 wool_sync_expression
2     : 'wool_sync' IDENTIFIER ;
```

**Figure 3.8:** The `SYNC` operation’s new syntax

---

\(^1\)This is a hassle which unfortunately is carried on to the new syntax also. The only workaround can be provided in runtime and it is a feature being developed.
1. wool_sync task1;
   SYNC (task1);
2. m = wool_sync task2;
   m = SYNC (task2);

Figure 3.9: Examples of usages of the SYNC operation.

### 3.5 The CALL operation

CALL is also responsible for instantiating a task. Its only difference to SPAWN is that the task is executed synchronously, thus immediately returning a value. Since a task is not a function per se, it can not be called as one. CALL is the specialized operation to make a task be called like any normal function\(^2\).

As a function call, CALL requires knowledge of the task identifier to be called and of course any arguments it may require. Again, both the old and new syntax are quite similar.

1 CALL ( name[,argument1[,...]] ) ;

Figure 3.10: The CALL operation’s original syntax.

1 wool_call_expression
2   : 'wool_call' IDENTIFIER ' ( expression )'
3   ;

Figure 3.11: The CALL operation’s new syntax

1. wool_call task1(n-1) ;
   CALL (task1, n-1) ;
2. wool_call task2( ) ;
   CALL (task2) ;
3. k = wool_call task2( ) ;
   k = CALL (task2) ;

Figure 3.12: Examples of usages of the CALL operation.

### 3.6 A parallel FOR loop definition

Probably the most complicated part of WOOL’s API is the definition of parallel loops. Also, it is the one construct which is very different between the original and new syntax. The original macro-based definition of such a loop construct, might seem very foreign at first. It is split into two distinct definitions. First is the

\(^2\)A feature version of WOOL would allow for all functions to act as tasks, thus negating the need for using the CALL operation and also having the wool prefix in their definition.
WOOL’s original API and WOOL C

The declaration of the loop, embedded in its place of execution; the second definition corresponds to the body of the loop and is defined in the global scope. Figure ?? provides the specifics of the syntax.

```c
FOR ( name,ival,lval,step[,argument1[,...]] );

LOOP_BODY_n ( name,COST,stype,sname[,type,name[,...]]) { ... }
```

Figure 3.13: The parallel FOR loop’s original syntax. The FOR macro is to be placed in the loops point of execution, while the LOOP_BODY has to be defined in the global scope.

WOOL treats a parallel for loop as a task to be executed multiple times, with the ability of spreading this execution among multiple threads. So its definition is a direct result of this definition, with LOOP_BODY indirectly being a task and FOR nothing more than an automated-multi-SPAWN. However, there are a few drawbacks with this implementation that are transferred to the user as extra overhead. First of all, separating the body of the loop from its declaration means that all knowledge specific to its declared scope is not shared. This is why, in defining a parallel loop, one has to keep track of all local variables used in its body, in order to pass them as arguments of the LOOP_BODY macro.

The LOOP_BODY macro follows the semantics of tasks. First argument is a unique identifier; COST is an internal constant [9]; stype and sname are the type and name of the step-identifier. The final part is a list of all free-variables present in the body, which act like arguments.

The FOR macro is much like the CALL, having as first argument the name and at the end the parameter list. In between these two, the arguments ival and lval are the initial-value and last-value of the loop respectively. The number of iterations is \(\frac{lval - ival}{step}\), rounded down to an integer.

In the new syntax, the parallel-FOR-loop is defined as a syntactically common but semantically restricted for-loop. It is syntactically common because it follows the exact same syntax as the regular for-loop of the C language. It is restricted in that all three parts of its declaration are allowed much less flexibility than a regular for-loop. For more explanations on this issue the reader is redirected [9] as it is not a purpose of this document to discuss how WOOL itself works internally. However, it should be mentioned that the original syntax expects the user to respect these restrictions in defining a for loop, although there is no mechanism for enforcing them, unlike the grammar of the new syntax which does.

As with the TASK definition, a FOR loop is prefixed with a special keyword to differentiate it from a normal loop. Other than that it is syntactically identical to a regular for-loop. The special declaration is used only to allow support for the special restrictions.

---

3Free variables, in respect to a specific scope, are all variables that are not declared in that scope. In wool we care for all but the global variables, as these can be available to the body; however, it is the developer’s responsibility to make sure that their declarations precede it.
A parallel FOR loop definition

```
1 wool_for_statement
2   : 'wool_for' '(' wool_for_init_statement wool_for_test_statement
3       wool_for_step_expression ')’ statement
4
5 int i;
6 wool_for( i = 10; i > 5; i-- ){
7 ...
8 }
```

**Figure 3.14:** The parallel-FOR-loop definition’s new syntax.

The initialization part is quite restricted as it does not allow for a declaration. The step-identifier has to be predefined in order to be accessible to every task generated from the loop. Hence its syntax is quite simple and straightforward.

```
1 wool_for_init_statement
2   : IDENTIFIER '=' expression ';'
3       | specifier_qualifier_list direct_declarator '=' initializer ';'
4
```

**Figure 3.15:** The syntax for a parallel FOR loop initialization definition.

Moving on, the test part is also restricted into using only four inequality operators. It is also necessary to point out that the IDENTIFIER is the step-identifier and it has to be on the left-hand side of the inequality.

```
1 wool_for_test_statement
2   : IDENTIFIER wool_for_test_op expression ';
3
4
5 wool_for_test_op
6   : '<'
7       | '>'
8       | '<='
9       | '>='
10
```

**Figure 3.16:** The syntax for a parallel FOR loop test definition.

Finally, the step part is no different, although it enforces constraints on the test part also. Specifically, if the step is decreasing the value of the step-identifier, then the test operator has to be that of either '>' or '>='. Similarly, '<' and '<=' are reserved for an increasing step value. This constraints are checked and acted upon in the back-end. It would be extremely complicated to enforce a syntactic constraint across multiple rules in the parser and so it was avoided. Moreover, the back-end has to recognize and analyse these definitions anyway, also providing much more flexibility.

Again, the IDENTIFIER represents the step-identifier and it has to be on the left-hand side of the expressions.
1 wool_for_step_expression
2     : '++' IDENTIFIER
3     | '---' IDENTIFIER
4     | IDENTIFIER '++'
5     | IDENTIFIER '---'
6     | IDENTIFIER '+=' expression
7     | IDENTIFIER '-=' expression
8     ;

Figure 3.17: The syntax for a parallel FOR loop step definition.
Front-end: A GNU C compatible parser

Almost any application developed in C uses libraries which are included to the project. Without external libraries a C application is not able to provide even the most necessary functionality such as an interface to input or output data. On most operating systems these libraries are distributed as part of the compiler. Especially for GCC, almost all of its libraries include a number of extensions to the ISO C standard. These extensions are part of the GNU C standard [24], a special version of the C language, implemented and supported mainly by the GNU C compiler, although others are fully compatible (ICC). Whether these features are sparse or dense in the code, the bottom line is that in order to parse a real-life C application a parser has to at least recognize and tokenize the relevant pieces of code, even if it blindly passes them on. Since this project aimed from the start to be functional in real-life application development, support for these libraries was deemed mandatory. This is the main reason behind having to implement a parser for the GNU version of the C language. Nevertheless, the supported extensions could also be used in any GNU C project.

In order to develop a GNU C parser one has to tackle a number of problems. The majority of the GNU C extensions either provide some standalone new functionality or modify existing ones. This fact has two major advantages for this project; first, making the parser understand ISO C is enough for the semantic analysis later on and second all support for GNU extensions can be simply added on top of an ISO C parser with minimal modifications necessary.

In short, the supported GNU C extensions are the following:

1. Built-in functions & types: these are not declared anywhere but recognized by GCC internally.
2. C preprocessor directives: not part of the actual C language, although used in almost every project.
3. Inline assembly: the `asm` operation, allowing a special recipe of assembly code for specific operations.
4. `__extension__` tags: keywords prefixing declarations which follow GNU syntax, incompatible with standard C.
5. Attributes for types and variables: construct following a non simple syntax that includes the `attribute` keyword and a list of other keywords enclose in double parenthesis. Attributes are hints to GCC on how to treat specific constructs.
6. Unnamed fields: when a structured, nested inside another structure, is not given a field name. Starting with GCC v4.4 all nested fields become accessible by the parent structure, as long as their field names are unique.
7. Alternative keywords: a set of alternative keywords to the built-in type-qualifiers, used to maintain consistency with legacy code.

4.1 Lexical analysis

The scanner was built using the GNU Flex program, hence the vocabulary was written according to the GNU Flex specification language [19] [20]. An initial vocabulary for ANSI C, as described in the ANSI C specification, was found [16] and modified to support ISO C. Appendix A on page 56 provides the final vocabulary used
for this project, separating the various extensions and modifications that were added in order to get a working vocabulary for the new WOOL-based language.

There is not much logic involved in the process of scanning except with the case of disambiguating between identifiers and user defined type names. These are both alphanumeric strings (symbols), that have been defined by the author of the program. Inside the definition of a new type, the type name symbol should return the token IDENTIFIER, but on every occasion later on (while this definition has not been shadowed) it is of token TYPE-NAME. Figure 4.1 gives an example of a scenario where things become complicated.

The C language allows for same symbol declarations with overlapping scopes, or as it is formally called *shadowing*. According to this technique, a symbol can be redefined in a nested scope. There can be no two declarations of the same symbol in the same scope however. So, a symbol that used to be a TYPE-NAME can later on be redeclared as a variable (IDENTIFIER) and vice versa.

```c
1  typedef int x;
2  typedef x z;
3  x foo(x x){
4       x = 2;
5  int y = 0;
6  x = y + 1;
7  return x;
8 }
```

Figure 4.1: A sample of shadowing that creates scanning conflicts. On line 1, *x* becomes an alias to the type *int*. This instance of symbol *x* should return the token IDENTIFIER. On line 2 though, *x* is used as a TYPE-NAME. On line 3, *x* is again used as a TYPE-NAME defining the same symbol as a variable. On line 3, the 1<sup>st</sup> and 2<sup>nd</sup> instances of *x* must return TYPE-NAME while the 3<sup>rd</sup> IDENTIFIER.

For the scanner to make the distinction it consults a symbol table that is created and maintained by the parser. This symbol-table, given a symbol, will return either the IDENTIFIER or the TYPE-NAME token according to the symbol’s most recent declaration, in a scope aware manner. When there is a conflict like in the parameter list of the function definition in figure 4.1, a rather complex mechanism takes care of the disambiguation. The scanner defines a flag variable (*defines*) that is set to true whenever there is a match of any type name (including user defined ones). If a symbol is matched while this flag is set to true then the token returned must be an IDENTIFIER. At that point the flag is set back to false.

The afore mentioned mechanism works because of the following rules for the declarations in the C language [15]:

1. Every declaration defines at least one IDENTIFIER.
2. A user cannot define a quantifier.
3. A user defined type cannot be combined with any quantifier<sup>1</sup>.
4. A symbol can be shadowed only once in one declaration.

Rules 1 to 3 tell us that if a user defined type is matched inside a declaration, there can only be one more symbol in that declaration and it has to be the IDENTIFIER. Rule 4 ensures that if multiple IDENTIFIERs are declared as a user defined type in one declaration, then only one of them can be of the same symbol as the type.

---

<sup>1</sup>*Quantifiers* are a virtual subset of the *type-specifiers* and should not be confused with *type-qualifiers*. Specifically this set includes long, short, signed and unsigned. These can be combined with other specifiers creating what are called *integral types*.
However, the conflict described in figure 4.1 is not the only one in the C language. There can be a situation where a symbol has been declared as TYPE-NAME while it is used as a structure tag. The appropriate token to return would be IDENTIFIER, even if the symbol table will tell otherwise. To that end, the scanner keeps track whether it is being active inside a structure declaration or not. This is achieved with a flag variable (instruct) set to true on encountering the STRUCT, ENUM or UNION keywords, which is subsequently set back to false after scanning the tag or the first opening brace if a tag is not present. This mechanism works because of the fact that the first symbol following the aforementioned keywords is always a structure tag [15].

Exceptions exist also in the case of a symbol following a structure field accessing operator, namely ‘.’ and ‘→’. This is handled by always keeping a buffer of the previous token which is checked when necessary to make the decision.

Furthermore, if a type definition is in place, one more flag variable (defines) takes a special value that makes the scanner disregard all exceptions and lookup all symbols in the symboltable. There are two possible scenarios in this case:

1. A symbol is a user defined type: the symboltable must have a single record of that symbol and it must be assigned to TYPE-NAME.
2. A symbol is the type being defined: the symbol-table has no record of that symbol so it defaults to IDENTIFIER.

This protocol will work because of the following facts of the C language:

1. Every type definition declares only one symbol, as a type-name [15].
2. All type definitions can exist only in the global scope.
3. The same symbol cannot be declared again in the same scope.

The above facts mean that using a declared symbol for defining a type is illegal in the C language. So, when a symbol is declared as a type, there can’t be any previous record of it in the symbol-table. On the contrary, after a symbol is declared as a type, it cannot be redeclared as anything else in the global scope again. This means that the only case where something can go wrong is when there is actually something wrong and this error has to be caught and raised appropriately.

A special case of the previous scenario is when the defined type is a structure declared inline with a type declaration (typedef). The declaration of a structure can include other declarations but most importantly it defines a nested scope so reusing declared symbols is allowed. The scanner will use its knowledge of when scanning a type definition in conjunction with the mechanism for identifying when inside structure definition bodies (the instruct flag), in order to handle the structure independently of the rest of the type definition.

Putting all of the above together, the complete protocol followed to figure out a user defined symbol’s token is as follows:

1. if the symbol is next to either ‘.’ or ‘→’ operators return IDENTIFIER.
2. if the instruct flag is set to true return IDENTIFIER and toggle the flag.
3. if the defines flag is set to true return IDENTIFIER and toggle the flag.
4. else do a lookup on the matched symbol.
5. if a record is found return its assigned token type. If it is a TYPE-NAME activate the defines flag.
Finally, in the cases of encountering a CONSTANT (a numerical or character string value) and of course any user defined symbol, the value itself is specifically saved for later retrieval by the parser. This is achieved with a special union structure that the scanner uses to pass extra information to the parser. This union is part of the Flex specification and the mechanism for handling it is automatically generated. This automatic mechanism is coupled with the parser. The structure has five members, although for describing the scanner we only need mention three of them. They are `intval` of type `int`, `floval` of type `float` and `strval` of type `string`. All values are saved in the corresponding structure member according to their type. String literals, hex numericals and identifiers all share the `strval` member. In all other cases of tokens, the `intval` member holds the numerical representation of the token.

### 4.1.1 Scanning GNU extensions

Extending the scanner is a very easy process. The only thing necessary is to enrich the vocabulary with the new words and statically assign tokens to them. As mentioned earlier the supported GNU extensions do not really interfere with existing syntax, rather add independent functionality. For that reason, adding the relevant words in the vocabulary introduced no conflicts or ambiguousness.

The C preprocessor directives, as explained earlier, are all but one discarded. `Include` directives are captured and passed on as a whole, meaning that there is no internal understanding of them. The scanner will capture the complete directive as one string field and pass it to the parser under a single token (`INCLUDE`). This feature is a remnant of past attempts to solve the multiple-inclusion issue, since the WOOL header file wasn’t initially split into its two parts. Finally, it wasn’t removed to allow a simple way of testing specific translation attributes of WPP, without using it through WCC. It is mentioned here because it is always possible for the user to invoke WPP as standalone leading to serious problems. For example, having include directives being extracted from inside ifndef directives will lead to including unwanted header files and possibly severe compilation issues.

Moving on, the `Inline assembly` instructions are captured and passed on as a whole. This is ok as far as the front-end does not need to understand them. Initially this was excluded from the supported features. `Inline assembly` instructions are quite tricky to capture as there is no specific and unique pattern that defines their ending. At first, a parenthesis counting mechanism was implemented. It used a simple integer variable as a counter. It started at 0 with the capturing of the `asm` or `__asm__` keywords and was increased by one for every opening parenthesis and decreased by one for every closing parenthesis. If the counter reached the number 0 again it would mean the end of the parenthesized block.

However, since such instructions spread among multiple lines, the scanner was incapable of keeping track of all transitions. Furthermore, moving these parts to the parser would require from it to syntactically fully understand the `inline assembly` and as mentioned earlier this was decided as beyond the scope of the project. So, the best solution came with exploiting one of Flex’s regular expression matching features; it follows the POSIX greedy algorithm by capturing the longest possible match. Exhaustive tests showed that the captured pattern was always the correct one, irrelevant to its complexity. The token returned is `ASM`, along with its definition stored as a string value.

---

2The `string` type is defined as part of the Abstract Syntax Tree, described in detail in a later section; for a complete reference of the AST see appendix C on page 79
Syntactic analysis

Figure 4.2: Scanning GNU inline assembly instructions.

Next, the __extension__ tag is captured and passed on as any other keyword. The assigned token is EXTENSION. Attributes are again a bit complicated however simpler than the inline assembly. This is because there is a terminating pattern to capture which is a double closing parenthesis. Combined with the exact same logic as with the inline assembly, a single Attribute instruction is captured as a whole and passed on as token ATTRIBUTE, along with its definition as a string value.

Figure 4.3: Scanning GNU attributes.

Unnamed fields do not require scanner support as it is a lossy relaxation of the ISO rules. Alternative keywords are supported also as simple keywords. A new token was added for each: __const is returned as token _CONST, __const__ as token _CONST_, __restrict as token _RESTRICT and finally __restrict__ as token _RESTRICT_.

4.1.2 Scanning the WOOL language

The new syntax introduced to implement the WOOL API is heavily based on existing ISO C vocabulary. Thus, the only new tokens required are the special keywords introduced to differentiate their respective WOOL constructs from similar pre-existing C ones. The relevant code abstract including the returned tokens is shown in figure 4.4 below.

Figure 4.4: Scanning the WOOL language.

4.2 Syntactic analysis

The parser was built using GNU Bison, hence the grammar was also written in the Bison specification language [8]. The final grammar is presented in Appendix B on page 62.

Syntactic analysis has two main goals. To syntactically verify the source program and transform it into a Syntax Tree while removing all unnecessary punctuation and other clutter. After the parser starts receiving tokens, it aggregates them until a sentence is matched. Then a new node is added to the tree representing this sentence. All the required information to recreate the sentence are saved inside that node. If for any reason a sentence is not matched, a syntactic error is raised.
A basic grammar for the ISO C language is provided directly by the C specification [3]. However, it is not production-ready since it produces at least one shift/reduce conflict, caused by the famous dangling else depicted in figure 4.5.

### 4.2.1 The dangling else

According to the sample in figure 4.5, there is no way for the parser to know to which if-statement the else belongs to. The specification defines the association clearly (§6.8.4.1):

An else is associated with the lexically nearest preceding if that is allowed by the syntax.

However, the corresponding grammar production does not define this association. Returning to the example of figure 4.5 one it is easy to see that when the parser reaches line 8 it will have two equally possible alternatives... Either to reduce the if-statement of line 7 according to the production on line 2, or shift again to match it with the production on line 3. Bison cannot guess the aforementioned association unless it is explicitly said to do so. This is very easy to do although it requires some tweaking.

```plaintext
1 if-statement
2    : IF '( expression ')' statement
3    | IF '( expression ')' statement ELSE statement
4      ;
5
6 if( expression1 )
7 if( expression2 ) statement1
8 else statement2
```

Figure 4.5: The grammatical rule for an if selection statement is given at the top. Below is a sample code that produces a conflict with that definition.

Figure 4.6 shows exactly how to solve this issue. It involves defining a new dummy token LOWER-THAN-ELSE with no association. A dummy token is expanded to nothing and can be matched arbitrarily as it represents void. Similarly, the token ELSE is defined as having no association. Then in the if-statement rule, the first expansion alternative is appended to include the new dummy token, with the %prec guideline that enforces lower precedence over the other alternatives. For that reason when the parser has the ELSE token in its lookahead it will prefer to shift than reduce as the second rule has higher precedence.

```plaintext
1 %nonassoc LOWER-THAN-ELSE
2 %nonassoc ELSE
3
4 if-statement
5    : IF '( expression ')' statement %prec LOWER-THAN-ELSE
6    | IF '( expression ')' statement ELSE statement
7      ;
```

Figure 4.6: Adding the rules at the top and modifying accordingly the first if-statement expansion alternative, tells Bison about the else association, as defined by the C specification.
4.2.2 C preprocessor directives

Most such directives are defined in the global scope and take up a whole line, or more. They are not mixed with C operations in any way, although it is allowed to have such directives between C language statements. In other words, these directives have the power to radically transform the source code by using a rather complicated mechanism. Once again, adding internal support for these directives would be pointless for this project thus the scanner does not handle them. On the contrary, upon scanning such a directive the scanner will abort with an appropriate error.

The single directive which is passed on to the parser is `include`. This directive except being used almost exclusively in the global scope, is also typically placed at the beginning of the file. This is why it is syntactically identified as an external declaration. Since there is no understanding of these directives, the parser places each instance of them on the tree as a single leaf node and moves on.

4.2.3 Inline assembly

The inline assembler GNU extension is actually a single token and a black box to the parser which according the grammar is treated as an `expression` and an `expression statement`. The later can be issued as any other statement, inside a compound block or as a standalone statement, with the corresponding rules shown in figure 4.7. The simple inline assembler expression though does not have all the privileges of any other expression. Actually it can be used only as a tail `expression` to a `declarator`; hence the only change to the grammar was to that rule alone as shown in figure 4.8.

```plaintext
1 asm_expression
2     : ASM
3     ;
4
5 asm_statement
6     : asm_expression ';'
7     ;

Figure 4.7: Syntactically supporting the inline assembler instructions. It can be an expression or as an expression-statement
```

```plaintext
1 declarator
2     : pointer direct_declarator asm_expression
3     | direct_declarator asm_expression
4     | ...
5     ;
6
7 statement
8     : ...
9     | asm_statement
10     | ...
11     ;

Figure 4.8: Syntactically using the inline assembler instructions. It can be used in a declarator or as an expression-statement
```
4.2.4 __extension__

Next the __extension__ GNU extension required also some modification of the grammar. This time the modifications were significantly more intrusive as the keyword can prefix any declaration except an enumeration. Thus two new grammar rules had to be added as shown in figure 4.9. The next step was to replace the usages of the declaration and struct-declaration rules with the new ones, wherever the new keyword is supported. This means everywhere in the grammar except from within a for-loop specification.

```plaintext
1 declaration-extended
2     : EXTENSION declaration
3       | declaration
4       ;
5
6 struct-declaration-extended
7     : EXTENSION struct-declaration
8       | struct-declaration
9       ;
```

Figure 4.9: Syntactically supporting the __extension__ GNU keyword. Both rules were added to expand the declaration and struct-declaration rules to support the keyword.

4.2.5 GNU Attributes

Similarly intrusive modifications were done for supporting the GNU attributes. Although Attributes are scanned and passed on as a black box, GNU C allows for comma-separated lists of them. This complicates the situation since such lists can expand into multiple lines and the scanner would be truly overwhelmed if it tried to capture them as one pattern. So, the list was created as a syntactic structure, by adding the logic to understand them. As figure 4.10 shows, two more rules where added. The attribute-specifier represents a single Attribute, possibly suffixed with a comma. This schema follows the same principles as with the dangling else. Next the attribute-specifier-list rule defines a right expanding list.

```plaintext
1 attribute-specifier
2     : ATTRIBUTE %prec DUMMY
3           | ATTRIBUTE ','
4           ;
5
6 attribute-specifier-list
7     : attribute-specifier
8           | attribute-specifier-list attribute-specifier
9           ;
```

Figure 4.10: Syntactically supporting GNU attributes. The first rule handles a ATTRIBUTE token as standalone or as part of a list. The second creates lists.

Attribute support was added in declarators, forcing the addition of one more rule that replaces the usages of the declarator wherever an attribute is allowed, as per the declaration rules. Structure specifiers and labeled statements complete the group of Attribute ready C constructs.
4.2.6 Unnamed fields

One more alteration done to the grammar was to support the unnamed fields GNU extension. According to it a structure (struct or union) can have a structure field that is assigned no identifier. Starting with version 4.4, GCC allows to access directly the fields of a nested structure, as long as they are recursively unique among all the fields in the parent structure. Figure 4.11 shows the grammar rule that was modified. The newly added token struct-or-union-specifier can actually be reduced to a specifier-qualifier-list, however in order to have a unnamed field it must be absent a declarator. The reason for the new rule to consist of a struct-or-union-specifier instead of a specifier-qualifier-list is that anything more general would allow for non structure fields to be unnamed also. Surely unwanted behaviour.

```
1 struct-declaration
2   : specifier-qualifier-list struct-declarator-list ';'
3   ;
4
5 struct-declaration
6   : specifier-qualifier-list struct-declarator-list ';'
7   | struct-or-union_specifier ';'
8   ;
```

Figure 4.11: Syntactically supporting the unnamed fields GNU extension. The top rule is the ISO version, demanding a declarator, while the bottom version supports the GNU extension.

4.2.7 Alternative keywords.

All of the alternative keywords defined in the GNU C standard are grammatically type-qualifiers. This grammatical non-terminal is nothing more than a group of such keywords, not defining any syntactic phrases. Moreover, these keywords can be used in a chain, interchangeable and it is a stated fact that they are idempotent in nature. The latter means that a repeated appearance of the same type-qualifier in a specifier-qualifier-list is identical to having a single appearance [3]. Thus the only alteration needed was to populate the aforementioned non-terminal with the new keywords as shown in figure 4.12 below.

```
1 type_qualifier
2   : CONST
3   | _CONST
4   | _CONST_
5   | RESTRICT
6   | _RESTRICT
7   | _RESTRICT_
8   | VOLATILE
9   ;
```

Figure 4.12: Syntactically supporting the GNU alternative keywords.

4.3 The parser’s symbol table

The most critical part for successful scanning and parsing of any GNU C program is the symbol table which is created and maintained by the parser, while used by the scanner. This table is complex enough to require its
own section in the report. As described earlier, the main purpose of the symbol table is to keep track of special alphanumeric strings, called symbols, in the source code. The entries in the table are used by the scanner in order to give context sensitive meaning to these symbols, ultimately allowing them to be assigned to the correct token. Figure 4.1 gives an example on how such symbols can have different meaning according to their context. Specifically, the same symbol might be assigned either the token IDENTIFIER or TYPE-NAME. A symbol is identified as one of these types according to the following rule:

A symbol is identified as a TYPE-NAME right after it has been used as an IDENTIFIER for a type definition, for all scopes equal or higher to the one of the declaration, until a new declaration, in a higher scope, uses the same symbol. In all other cases a symbol is identified as an IDENTIFIER.

The symbol table and its entries must abide by the following properties:

1. A string might have multiple entries in the table.
2. Entries must be unambiguously defined as either an IDENTIFIER or a TYPE-NAME at any given time.
3. The table must keep its entries in a scope aware manner.
4. For each scope there can be only one entry of each symbol.
5. Upon lookup the table must return the most recent entry, matching the given symbol.

Before explaining how these properties came to be implemented there has to be a reference on the exact implementation of the symbol-table. The main structure defined is called the symtab and it is a table structure. For a definition of this structure please look in appendix D on page 83. Symtab’s cells define three fields: type, word and scope. The first is an enumeration of the two possible tokens (IDENTIFIER or TYPE-NAME) plus a third (NOTHING) used internally and explained later. Word is a copy of the symbol’s string. Scope is the scope where the corresponding symbol was defined in. The rows are identified by a hash of the word field. Thus, symtab defines buckets of symbols, each having the same hash. New cells are added at the head of each bucket.

Aside from symtab, symbol-table defines and uses one more table. This is called symind and it is a reference table of the hash-keys in use in the symtab per scope. More specifically, rows are identified by scope and the cells are a list of row keys (hashes of the word fields) that have a symbol entry for that scope. Symind provides a huge performance benefit as it allows to know exactly which rows to visit when trying to find symbol entries of a specific scope. Also, since symtab’s rows are implemented as stacks (LIFO), these symbols are going to be consecutive. Furthermore it was proven that the compiler needs not to ever search for a scope other than the current active one, thus this consecutive queue of symbols is going to be at the beginning of the row.

Finally, the symbol-table is completed by the use of a stack (see appendix E on page 85) that holds extended information for the scopes. This stack also is used to keep track of initialization and termination of scopes. Mainly it defines the current scope as the last entry pushed onto the stack. Furthermore, each entry keeps some scanner state information. When a scope is initialized the scanner’s state changes in many occasions. This state is stored in this stack thus when a nested scope is terminated, the parent scope’s state can be resumed.

The algorithm is quite straightforward... whenever a symbol entry is encountered it is entered in the symbol-table followed by two arguments, the scope and its type. The symbol is saved at the top of the row identified by its hash. This way a word can be stored in the symbol-table multiple times, which obeys property 1. If symind has no entry for this row-key in this scope, then one is created. When a scope is terminated, the symbol-table

---

3The term most recent is actually used literally. Since when a scope is terminated, all entries in the symbol-table associated with it are removed, the most recent entry is the latest declaration of a symbol in either the current scope or one of its parents.
uses the *symind* references to find the rows in the *symtab* that have entries for that scope and removes them. If a new scope is entered, the algorithm stays the same with the *symind* using a new row, identified by the new scope number. This process as a whole qualifies for the requirements of property 3.

When there is lookup upon the symbol-table, a hash of the requested symbol is taken and the corresponding row in the *symtab* is exhaustively searched from the top for an exact match. If one is found it is returned; if not **NULL** is returned. Since symbols are prepended to each row, what is going to be returned is the most recent entry, thus implementing property 5.

Properties 2 and 4 are quite more complicated and are implemented in the parser and scanner. As discussed many times earlier, a symbol declared as a type is assigned the token **IDENTIFIER** when used in the actual declaration and this is handled without consulting the symbol-table. However, what is important is to then store that symbol as a **TYPE-NAME**. In order to accomplish this we need to use a few facts:

- Entries to the symbol-table should be added only during the corresponding symbol’s declaration.
- A type definition declares exactly one **IDENTIFIER** as **TYPE-NAME** [15].
- The **IDENTIFIER** token for all declarations is derived from a unique grammatical rule [15].

According to these, there is only a single grammatical rule that needs to add entries to the symbol-table (see fig. 4.14). Also, for each declaration this rule is going to be matched exactly once. Finally, it should always add symbols as **IDENTIFIERs** except if the underlying declaration is a type definition, in other words the parser has to figure out the single case where a symbol is a type-name and fallback to identifier in any other. This last point however is where things become really complicated. Imagine a structure being defined as a type. This is a pretty common scenario in C. Imagine that the structure definition is inside the type definition. The bottom line is that it is very difficult for the parser to know which of all the declared **IDENTIFIERs** will be the actual type-name, since it has to parse the complete type definition first.
Since backtracking is not a welcome feature in this parser, an indirect method was devised to achieve the goal. To make it more clear let's take a step back and remember that the scanner has been set up to look for a few exceptions before consulting the symbol-table. In short these are symbols directly following structure field access operators (either '.' or '->') or the STRUCT, UNION and ENUM keywords; symbols following a list of integral types or a user defined type. All these special cases default to identifier with no consideration to their context. This means that there is no need for the symbol-table to keep track of these symbols as it will never be consulted for them. Going one more step forward, it becomes clear that inside a type definition, however complicated, the only symbol that has to be stored is the actual type-name being defined. To generalize, the same applies for the fields of all structures and the arguments of function prototypes.

One more optimization concerns the function prototypes or in other words a abstract-direct-declarator followed with a parenthesised parameter-type-list. The latter is nothing than a list of function parameter declarations, which can be either only a type or a complete declaration. Since a prototype has no body, these symbols effect no scope hence they do not require to be stored either. However, a function prototype can also be used to define a new type so it is not something that can be blindly skipped. The solution found was having a flag variable denote whether the scanner is inside a parameter list, allowing it to skip what is not needed. This flag is called in-par-list and it is set to true right after the opening parenthesis of a parameter list. Obviously it is set back to false before the corresponding closing parenthesis (see fig. 4.15).

```c
1 direct_abstract_declarator
2     : ...
3     | ...
4     | direct_abstract_declarator '(' parlist_enter
5     parameter_type_list parlist_leave ')'
6     ;
7
8 parlist_enter
9     : scope_enter { in_par_list = 1; }
10    ;
11
12 parlist_leave
13     : scope_leave { in_par_list = 0; }
14    ;
```

**Figure 4.15:** Keeping track of parameter lists for the symbol-table. A special flag is activated whenever the parser goes inside a parameter-type-list.
Finally, the parser uses one more flag variable called is-typedef to denote if the current declaration is actually a type definition. This is set to true whenever the `TYPEDEF` token is encountered; it is set back to false after storing the corresponding `TYPE-NAME` in the symbol-table.

To sum everything up, the final algorithm is quite plain and short. As a direct result of this algorithm property 2 is implemented.

1. When matching an `IDENTIFIER` inside the declaration, if is-typedef is true AND `in-par-list` is not, the symbol is kept in a buffer, else stored as an `IDENTIFIER`.

2. At the end of parsing a declaration, if is-typedef AND a buffered symbol exists store it as a `TYPE-NAME`; set is-typedef to false and clear the symbol buffer.

As for property 4, it is much more complicated as there are exceptions to be considered. An amended version of it is being handled in the back-end because in order to enforce such a constraint the compiler needs to know extensive contextual information. An example of an exception would be prototypes of the same function or forward declarations of the same structure; these are allowed to exist multiple times in the global scope as long as they are identical. An even more simpler example would be a prototype and the corresponding function definition which both declare the same symbol in the same scope.

### 4.4 Identifying and handling scope

All through this chapter there has been talk about scopes. These, often called environments, attach a special meaning to various instructions in the code and have to be carefully recognized and handled. First let’s see where are these encountered; figure 4.1 provides the grammatical rules where scopes are initialized and terminated. The grammar manages to capture these events by issuing dummy tokens in specific places in these rules. These tokens are listed in figure 4.2. Whenever, such a token is matched its assigned semantic action is executed, performing the desired effect.

<table>
<thead>
<tr>
<th>direct-abstract-declarator</th>
</tr>
</thead>
<tbody>
<tr>
<td>direct-declarator</td>
</tr>
<tr>
<td>function-definition</td>
</tr>
<tr>
<td>struct-or-union-specifier</td>
</tr>
<tr>
<td>enum-specifier</td>
</tr>
</tbody>
</table>

**Table 4.1:** Non-terminals whose productions perform scope initialization and termination.

The dummy tokens can be split into two categories... `scope-enter` and `scope-leave` directly perform initialization and termination of a scope respectively. These tokens expand to an empty production meaning that they correspond to nothing. These tokens can be matched arbitrarily with one exception. If used in a production as following a non-terminal token, then the scanner will not appropriately match that rule as dummy tokens carry reduced priority to other rules that might provide the expected terminal token instead. So, in cases where a rule expands to multiple alternative productions that have the same sequence of tokens up to the point of insertion of the dummy token, it must be inserted right after a terminal symbol. If no such similarity exists between productions the problem does not exist.

Special mention should be given in the mechanism controlling the scope in function definitions. This seems asymmetric as there are two scope initialization tokens and only one for termination. The reason is that func-
tions actually create two scopes, one nested inside the other. The outer scope is defined right after the function identifier symbol and before the parameter list. This action activates a flag (in-funcpar) to keep track of currently parsing the function parameter list. The second scope is initialized by the function body, this again activates another flag (in-function). Scope termination for both happens at the end of the function body.

<table>
<thead>
<tr>
<th>funcpar-enter</th>
<th>function-leave</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct-enter</td>
<td>struct-leave</td>
</tr>
<tr>
<td>parlist-enter</td>
<td>parlist-leave</td>
</tr>
<tr>
<td>scope-enter</td>
<td>scope-leave</td>
</tr>
</tbody>
</table>

Table 4.2: Dummy non-terminals used to initialize (left column) and terminate (right column) scopes. A scope can be terminated by the non-terminal of the same row to the one that initialized it.

The reason for the complexity of this mechanism is a direct consequence of complexity derived by the C language definition schema and more specifically declaration prototypes and functions without parameters. In short the practical reasons revolve around the same grammatical rules being used to match both an prototype-function-declaration (where parameter identifiers are given) and a full definition. In the case of a prototype there is no function body to capture the end of the function and terminate the scopes. In the case of a full function definition the scope of the parameter list has to be carried across the whole function body; otherwise the symbols of the parameters would not be in the table when used inside the function. Figure 4.16 shows the termination of the function prototype’s parameter-list scope.

```
1 declaration
2     : declaration_specifiers init_declarator_list ’;’
3     | ... 
4     ;
```

Figure 4.16: Terminating the scope of a function prototype’s parameter-list.

4.4.1 Built-in functions & types

One GNU extension that was supported in a minimal way are various built-in functions & types. Minimal because only the __builtin_va_list type was actually supported. However, the mechanism was put in place to add support for more in a fairly easy and straightforward way.

However, what are these functions and types? They are actually type-name symbols and function-definitions that are recognised internally in the GNU C compiler without providing an actual declaration for them. That way a parser has no way of acquiring an entry for them in its symbol-table. To provide support it is only necessary to add this entry manually, with type TYPE-NAME for types and IDENTIFIER for the function-identifier symbols. Also doing that in the global scope would ensure that these records remain in the table all the way through processing a source file.
4.5 Intermediate representation

The next step to parsing is semantic analysis. This is a context sensitive interpretation of the source program combined with transformations and optimizations. However, while the program is in the form of text, it is not very usable or even traversable; much less semantically analysable. This is why it is imperative to transform the text into a more usable form. This form must allow:

- quick and reliable traversal of the whole program,
- random direct access to any of its components,
- unambiguous extraction of the semantic nature of each instruction,
- minimization of the amount of stored data.

The first two properties are met with the use of a tree structure, a directed acyclic graph (DAG) where each node has at most one parent. For the third property there has to be an injective correlation between nodes and semantic concepts. Finally, for the minimization of data, the aforementioned correlation has to be unequivocal in a way that the only data saved would be those that differentiate nodes of the same semantic concepts.

In contrast to common compiler practices [5], there is no parser tree being created. Instead, the parser creates directly an Abstract Syntax Tree. Since the back-end requires to contextually recognize only a small portion of the C language, there is no need having a clear representation of everything else. This reduces the complexity enormously allowing to move directly into an AST. Of course a line has to be drawn here in order to clarify that the AST used is actually a middle-ground solution between a Concrete Syntax Tree\(^4\) and a truly AST which however scales radically towards the latter.

Two key aspects of the AST is first being able to create lists of identical nodes and second to allow reflection\(^5\) of nodes. The latter is necessary for meeting property 1 for reliable traversal since it wouldn’t require special knowledge on the semantic type of the next node in order to advance. When traversing a node it should be able to tell by itself what it represents and how to be processed. A significant outcome of implementing reflection also, is that it indirectly solves the problem of creating lists.

```
1 struct _AST_root {
2   _AST_ left;
3   _AST_root right;
4 }; struct _AST_ {
5   _AST_pos pos;
6   _AST_type type;
7   void *ast;
8 }; struct _AST_seq {
9   _AST_seq_type type;
10  _AST_ left;
11  _AST_ right;
12 }; Figure 4.17: The fundamental structures of the AST.
```

---

\(^4\)A Concrete Syntax Tree is the most elaborate tree model view of the source code possible. It defines a one to one correspondence between source operations and tree nodes, without striping anything more than unnecessary punctuation.

\(^5\)Reflection is the ability of an entity to observe and modify its own structure and behaviour [23].
The AST is divided into three fundamental node types. These are shown in detail in figure 4.17. Structure _AST_ is the basic building block of the AST; it is used for every node in the tree while it works as a wrapper class for storing the actual information. Having a generic class wrapping around every actual content, holding the appropriate properties that uniquely characterize’s it, is exactly what allows for the node to implement reflection. _AST_ nodes can be traversed blindly of any type considerations, while at any given moment providing clues on how to actually parse their content. This form allows total separation of the language semantics from the handling mechanism of the tree.

_AST_seq_ is a special structure that is responsible for creating lists of _AST_ nodes; it is important to clarify that _AST_seq_ is not the only way to assign multiple children in an _AST_ node. On the contrary it is used exclusively to allow a variable number of children. However, they cannot be inserted in the tree directly, without being wrapped into a _AST_ first, in other words _AST_seq_ can be the child of an _AST_ node but not of a _AST_seq_ node. Also, even though a _AST_seq_ structure does not represent an actual node, some of the different types encapsulate printable information that has been striped out, like various delimiters (i.e. commas); hence the role of the wrapper _AST_ node is to signify this information.

Finally, _AST_root_ wraps around external declarations simultaneously connecting with the next external declaration. This chain of _AST_root_ instances represents the backbone of the source program as shown in figure 4.18. The reason behind this special structure is very important and not at all a design decision. It actually solves a great problem that would normally be inherited to the AST from the way the parser works.

In the C grammar, external-declarations are chained together using the translation-unit grammatical rule. According to the second production (fig. 4.19) a new external-declaration is appended to the tree (instead of being prepended). This is to be expected since the parser follows a top to bottom path of parsing. However, the semantic actions that create the AST work according to the following procedure...

When a grammatical rule is matched (telling us that all of its right-hand-side tokens have already been parsed), a new _AST_ structure is created representing the symbol in its left-hand-side. The latter is then forwarded as a right-hand-side token at some next reduction.
The above algorithm applied to the translation-unit chaining rule, would really mess up the ordering of the external-declaration tokens. This is because the new _AST_root structure that is created for every declaration will become a right-hand-side non-terminal in the next application of the translation-unit rule; as such it will be considered as nested inside that future declaration... Constant application of this rule would eventually reverse the order of the declarations. This would be catastrophic as C demands for every used symbol to have been previously declared.

The only solution is breaking the chain by switching places between the right and left-hand-side non-terminals. In other words embedding each new declaration as a child node to its predecessor and then forwarding the new declaration. Nodes in the tree have no knowledge of their parents but know exactly which are their children. By using the above mechanism, "parenthood" information will be unknown for all the structures that take place in every translation-unit reduction. However, as it will be shown later on, this is of absolutely no importance, since traversal is always unidirectional from parent to children.

Moreover, the reason why _AST_seq would not be suitable to be used in the place of _AST_root is simply a matter of efficiency and clarity of the code. _AST_root provides internal chaining and can work as a standalone structure node. _AST_seq on the other hand, has to be wrapped inside _AST_ nodes which in total would require a much more memory than really needed. The only benefit would be to have less structures to deal with, thus simplifying traversal. However, as shown previously, the totally different nature of the external-declarations chain, renders this benefit negligible as there would be necessary to implement special treatment anyway.

1 translation_unit
2       : external_declaration
3       | translation_unit external_declaration
4       ;

Figure 4.19: translation-unit grammatical rule.
Figure 4.20: An abstract sample of a tree which includes a list. An example of such nodes could be multiple *statements* which make up one *compound-statement*.
5

Back-end: Analysis and transformations

![Diagram of back-end process]

Figure 5.1: The three-part process of a back-end.

The back-end part of the compiler is a virtual grouping of specific processes that follow the front-end's analysis. The difference is that the back-end performs processing that is targeted to the specific purpose of the compiler, while parsing in the front-end is quite generic and similar across different languages.

The back-end performs the following specific tasks:

1. Analyse and understand the **WOOL**-related pieces of the program.
2. Extract all the information required for the target language transformations.
3. Perform the transformations.
4. Output (print) the produced target program.

All of the above tasks but the last are simultaneously performed by one single-pass mechanism. As stated many times before, C is an imperative language that requires strict ordering of declarations. Therefore, when a specific part of the source program needs to be analysed and transformed, all the required information to do so must have already been processed and stored [4]. The main idea behind this mechanism is going through the whole tree which goes against the initial intuition.

Since **WOOL**-related code might be scarce in the tree, it is expensive to parse the whole tree, both in memory and time. Instead there could be a mechanism to create a reference table of such parts during a previous step, ideally while appending these nodes to the tree. Then it would be simple to visit these nodes directly, transforming them accordingly. However, this quickly proved to be more of a headache than an improvement. One of the most important pieces of information needed are the free variables of the scope defined by the **WOOL** constructs, namely **TASKS** and **FOR** loops. Acquiring this information requires backtracking through the whole tree preceding the node in question. Even if this was done intelligently, avoiding repeated traversals of the same nodes, the benefit would be minimal. The negating factors are two. First of all, in order to perform backtracking nodes would have to know their parents, information which would heavily increase the memory footprint of the tree. Also, it is obvious that the whole tree up to the last **WOOL** node would have to be parsed in all cases.

This model would benefit the compiler only in execution time and exclusively when the subtrees following the last **WOOL** node are significant in length. However, the main concern for a compiler is memory consumption. It is of much less importance how fast the compiler is if it needs excessive amounts of memory to run. Hence, the lighter model of no backtracking and full parsing of the tree was selected. The following sections try to decompose the back-end in a way that each chapter focuses on one task. However, the reader should always keep in mind that much of the operations of the back-end happen in a mixed order.
5.1 Code transformations

Transformations concern the specific WOOL extensions of the source language. These are five added constructs as explained in chapter 3. When talking about transformations, there are two distinct operations being performed, non simultaneously. Actual transformations happen during the final step of printing. However, there has to be extensive semantic analysis done beforehand which modifies specific nodes of the tree by adding or removing stored information, or even moving the nodes themselves around the tree.

Especially the nodes of the three operations, namely SPAWN CALL and SYNC, are not subject to any modification during semantic analysis. Since the source and target language hold the same information but different syntax, transformation happens altogether on the printing level.

This is not the case however with TASK definitions and FOR loops. The first needs to acquire an accounting of its parameters plus a new syntax. The information is collected during the analysis step, while it is printed directly using the new syntax in the next step. The latter needs to know of all free variables in its body plus being split into two parts, a new task representing the body of the loop extracted to the global scope, plus a call to this task where the initial loop used to be. These operations are spread during both phases. During analysis, the node is split with the body extracted from the tree completely. Also, all new information is collected and/or calculated. It is not before printing however that the new loop-body task is injected back to the tree simultaneously being printed.

One common operation during this processing, is error detection; it is happening exhaustively during the back-end’s semantic analysis.

5.2 Semantic Analysis revisited

Semantic analysis in the back-end revolves around traversing the abstract syntax tree and proactively extracting and storing specific information to allow for transforming pieces of code to the target language. The information needed is all the declarations of all the symbols, including functions, parameters and variables. By referring to declarations instead of just symbols, it means including the symbol’s full type specification; gathering the type specifications is actually the most significant part of the semantic analysis. To achieve this, the compiler uses a series of node handler functions. Each of these handles specific groups of nodes of the tree. The separation of the nodes is of course based on their content (for a clear overview see appendix C on page 79). These functions work recursively, calling one an other.

A problematic area in this mechanism is that sometimes nodes of the same content have different meaning according to their context. This is why a centralized, table-of-contents like, function would not work. Splitting the handling logic into multiple specialized functions was necessary as anything else would require a number of parameters to keep track of context that would be much less flexible in future customization. However, the number of these function was kept to the minimum of one function per conceptual group of language constructs.

Once again this analysis step makes use of a complicated table structure to store, organize and retrieve records of interest, in particular symbol declarations. This table is called DC-table and while it resembles the symboltable quite a bit, it actually is significantly extended and more complicated. A section has been dedicated on exactly how it works internally although a thorough explanation of its API has to be given here.

The most important notion to be understood is the nestability of declarations. There can be a structure declared as a type, with the declaration of the structure embedded into the type definition. However, the same structure
can be used again, as if it had been declared alone. So while traversing a declaration it is possible to have to pause in order to traverse a new declaration, resuming afterwards. Nevertheless, there is also the possibility for two symbol declarations to happen simultaneously (i.e. declaring multiple variables of the same type, using a comma delimited identifier list). In that case, the underlying mechanism has to create two or more records in the DC-table keeping some, but not all, fields the same. Finally, there has to be given great significance to the fact that declarations are composed of a complicated hierarchy of distinct multiple nodes, hence they are almost all handled by different functions. A declaration consists of two to three of the basic conceptual building blocks of the C language. So, one more obstacle is the necessity to have a mechanism that can work asynchronously and incoherently in a totally arbitrary way.

The final solution was based on one major extension over the symboltable. This included two temporary stacks. When new entries are initialized they are pushed on top of the first stack and remain open. This stack is called temp. In order to move this temporary entry to the table it has to be finalized. This feature allows for declarations to be captured and stored in multiple steps, even achieving the desired functionality of pausing capturing of one entry in order to focus on a nested one.

```
1  void foo(int i, char *b){
2    char a, *b;
3  }
```

Figure 5.2: Examples of nested and multi-symbol declarations. Parsing the declaration of foo would have to be paused to parse its arguments. Then paused again to parse the local variables, which is a multi-symbol declaration. When a is completed, a new declaration will be initiated for b which will automatically be assigned the type of a plus the pointer.

The second stack in the DC-table is called store and is used to move nodes to the global scope. These nodes are removed from their position on the tree and put onto the stack. Later they will be injected back to the tree in the global scope. An example of a scenario using this mechanism are the WOOL FOR loops. Such a construct requires to be split into two parts, with the second being a specialized task definition. As such it requires to be moved from its non-global scope to the global scope. Once again there are nestability issues. FOR loops can be nested; this means that their respective extracted bodies must be injected in the right order... it is the outer loop that calls the newer one, thus the declarations of their bodies must be injected in reversed order. Using a stack solves this problem since according to the LIFO algorithm the newest loop body will be injected first defining it before the "older" loop body tries to spawn or call it.

One aspect where the DC-table is less complicated than the symboltable is scope handling. It is reduced to keeping track of the current scope. Since the tree has been produced, each language construct is represented with its own subtree. So, the only use of scope awareness is to understand the current position and of course to enforce some priority ordering amongst the records.

The traversal mechanism always interacts with the last entry in the temp stack. As it goes through different nodes, it blindly stores its corresponding information to that entry. This logic allows for pausing the traversal of a declaration in favour of a new, nested one. When a new declaration is found, a new entry will be pushed on top of the stack. When traversal is finished, it will be moved into the table, making the previous entry the current one; this allows the mechanism to continue flawlessly. In the situation of multiple declarations, a new entry will be created upon traversing the next symbol. The mechanism is aware of such situation so it will retrieve the common type specification and copy it to the new entry (appending qualifiers and pointers as needed).

A case in need of special handling is with forward declarations (prototypes). Such declarations are traversed
and stored normally. When a symbol which might have a prototype is traversed (structures and functions), it is
looked up in the DC-table and if found then its entry is used and appended with the new information, instead
of creating a new entry. This way there are no double entries for the same symbols.

5.3 Target language printing

Printing happens in a similar manner to analysis. It traverses the whole tree, amending any outstanding nodes
in the store stack along the way and printing everything to a buffer. At the end this buffer is outputted to a
new file. This file has the same base name as the input file but a .c extension. In case of the input file having
the same filename, its extension is replaced with .new.c. This schema guarantees a unique filename since the
replaced text contains only one dot character, in contrast to the new text that has two.

The printing process is not really formatting the final code any more than it is necessary to avoid errors. A
special aspect of printing is reintroducing punctuation. This includes spaces which in many cases are of fatal
significance, for example consequently printing two custom defined symbols (i.e. type and identifier). If a
space is not included in between then they will become one string. However, all custom symbols are handled
by the same printer function. So, one might think that adding a single space after each symbol is enough; how
about the situation where the symbol follows a qualifier? A solution would be to add a space before the symbol
also; then what about accessing structure fields, where the symbol must directly follow a '.' or '->' operators?
The only solution to this problem is to handle all punctuation and spacing inside the most abstract nodes. It is
these nodes which have knowledge of the context of all of their components.

```c
/* merging symbols */
typedef _int_c _int_c_p;
typedef _int_c_int_c_p;
typedef_int_c_int_c_p;

/* splitting expressions into two */
s-> b=1; p. a = 0
```

Figure 5.3: Samples of printing fallacies and pitfalls.

5.4 The declarations table

A basic description of how the DC-table is structured and its operations has already been given in a previous
section. This section will go a bit further taking a thorough look on how exactly the DC-table’s functionality is
accomplished.

To recapitulate the DC-table is comprised of two TAB-tables just like the symboltable. In addition there are two
stacks, the first keeps all new entries as temporary until they are characterized as finalized. The second keeps
nodes that need to be extracted to the global scope. Finally, scope handling is minimized to keeping track of
the current scope while performed semi-automatically in conjunction to other functions.

Operations can be divided into three sets of functions. The first set handles the most important functionality
of initializing the table, adding and removing entries as well as lookups. It is important to note the role of the
temp stack and how it gets involved in most of all the functions of the table. When a new entry is added to the
The declarations table it is pushed on top of the stack. This procedure cannot be overridden. While an entry is inside the stack and more specifically at its head, it is considered active or the current record of the whole DC-table.

While an entry is active it can be accessed and modified directly. The second set of functions performs automation of the most common tasks and modifications against the active record. Since declarations do not overlap in C, it is very convenient that at any given point the active record represents what is being captured by the traversal mechanism also. In other words lookups are used only for using a finalized record, since all data entry is done exclusively with the active record.

As hinted earlier, in order to move an entry to the permanent store of the table, it has to be characterized as finalized. The corresponding function in the API will take care of both moving the active record as well as restoring the next one as active.

Scope is handled semi-automatically in the DC-table. This means two things. First of all to initiate a new scope the corresponding function has to be called. This will perform all internal configuration for changing the state to a new scope. What is totally automatic though is assigning a record to a scope since it will be assigned to the current one. Terminating a scope also requires calling a function of the API, although this function will take care of removing all related records from the table.

All table lookups are executed against the permanent scope in the same manner as with the symboltable. However, a temporary record is as much "active" as any other. For example, the declaration of a structure which declares a field that has a type of the structure itself; when declaring the field, the structure declaration is still not-finalized although being used. Normally, the DC-table would have to be able to lookup non finalized entries. However, this feature is totally useless for the purposes of this compiler so it was not implemented. No semantic analysis is done during type definitions but only of their usage for declaring new symbols.
Evaluation

Evaluating a compiler like the one in this project has no merit outside the small boundaries of the following question: "Does it work?". A compiler is meant to be executed without significant performance considerations since compilation is generally accepted as a slow process. This of course does not mean that performance is of no consideration at all, rather that it is secondary to other aspects of the process. Those being the quality of the generated code and the memory being used.

6.0.1 Generated code quality

Since the source language is just a superset of the target language, code generation is quite limited. So, to investigate the quality of the outputted code, it has to be split into two parts. First is the unaltered code and how well is it passed through the compiler to the output. Second of course, is the generated code and how complete and error-free it is.

Starting with the generated code, a formal specification of the translation is needed, so it is presented in figures 6.1 and 6.2. For simplifying this specification, the translation process has been represented by four conceptual functions. The first one, \( \text{Tr}_0 \), performs the actual translation from WOOL C to GNU C. It accepts any one part of the language as parameter and assumes an environment \( \varrho \); it outputs the translated code as one or two values. Next, \( \text{TrGlob} \), accepts a LOOP_BODY construct as parameter and it moves it to the global scope right before the start of the highest non-global scope, which is parent to the initial \( \text{wool_for} \) construct. Finally, there are two auxiliary functions; \( Df[] \) accepts one symbol as parameter and returns a boolean value whether the symbol is declared in a given environment; \( \text{Inv}[] \) accepts one expression as parameter and returns a boolean value whether it is invariant in its scope (and all nested).

It is assumed that the translation function does not modify any non WOOL C explicit syntax. One exception to this rule is with code nested in such syntactic constructs.

\[
\text{Tr}_0[\text{wool} T f(T_1 e_1, \ldots, T_n e_n)] \varrho = \text{TASK}_n(T, f, T_1, e_1, \ldots, T_n, e_n) ds', \emptyset
\]

where \( ds' = \text{Tr}_0[s] \varrho \)

\[
\text{Tr}_0[\text{wool}_\text{spawn} f(e_1, \ldots, e_n)] \varrho = \text{SPAWN}(f, e_1, \ldots, e_n), \emptyset
\]

assert \( Df[f] \varrho \)

\[
\text{Tr}_0[\text{wool}_\text{sync} f] \varrho = \text{SYNC}(f), \emptyset
\]

assert \( Df[f] \varrho \)

\[
\text{Tr}_0[\text{wool}_\text{call} f(e_1, \ldots, e_n)] \varrho = \text{CALL}(f, e_1, \ldots, e_n), \emptyset
\]

assert \( Df[f] \varrho \)

Figure 6.1: Formal specification of the code translation process, part I
\[ \text{Tr}[\text{wool}_\text{for}(T \ i = e_1; \ E_2; \ E_3) \triangleright \varrho = \text{FOR}_n(f, e_1, e_2, t, v_1, \ldots, v_n), \ ds'] \]

where \( f \) is a new unique symbol

\[ E_2 = \begin{cases} 
LT = \begin{cases} 
i < e_2 \\
i <= e_2 
\end{cases} \\
GT = \begin{cases} 
i > e_2 \\
i >= e_2 
\end{cases} 
\end{cases} \;
\]

\[ E_3 = \begin{cases} 
Inc = \begin{cases} 
++i \\
i++ \\
i += e_s 
\end{cases} \;
\end{cases} \;
\]

\[ Dec = \begin{cases} 
- - i \\
i-- \\
i -= e_s 
\end{cases} \;
\]

\[ t = \begin{cases} 
1 \text{ when } E_3 \in \text{Un} \\
e_s \text{ when } E_3 \notin \text{Un} 
\end{cases} \;
\]

\[ d_{\text{new}} = \text{LOOP}_\text{BODY}_n(f, 100, T, i, T_1, v_1, \ldots, T_n, v_n), \ s' \]

\[ ds, s' = \text{Tr}[s][i \mapsto T \ i] \]

\[ ds' = \text{TrGlob}[d_{\text{new}}]; ds \]

\textbf{assert} \ E_2 \in \text{LT} \land E_3 \in \text{Dec} \\
E_2 \in \text{GT} \land E_3 \in \text{Inc} \\
\text{Inv}[e_2] \\
\text{Inv}[e_s] \\
\]

\[ \text{Tr}[e]\varrho = e, \emptyset \]

where \( e \in \text{GNU C} \)

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image}
\caption{Formal specification of the code translation process, part II}
\end{figure}

Since most of the explicit syntax in WOOL C is simply a collection of keywords or invariant code (in respect to the translation process), the only parts of the algorithm in need of verification are listed below:

1. Counting and identifying the task arguments.
2. Identifying a void returning task.
3. Verifying a spawned, synced or called task exists.
5. Identifying free variables in the body of parallel loops.
6. Checking the invariance of a parallel loop’s test condition right hand side.

Translating any of the spawn, sync or call operations requires only a syntactic restructuring of the information already present in each one. Furthermore, semantic validation of them is successful since C requires all declarations to precede their uses; this means that the given environment to the translation will include them.
Analysing a task definition is quite straightforward also. All information required to perform and the translation is included in the definition itself. Counting the task arguments is trivial and executed simultaneously with splitting their declarations into two parts.

The only part of the translation process that is of interest is the parallel loops. The many constraints applied here and the extent to the transformation, increase the complexity of this procedure dramatically. Figure 6.2 shows a number of the checks being performed during this process. The most important issue comes with the extraction of the loop body as a separate task and placed in the global scope. The question that comes to mind involves the symbols being used in that code block and their scope, being consistent with the initial intention of the original code. However, the validity of the transformation can be proved with a few simple arguments. As said before it is trivial for the analyser to know which symbols are declared in a parent scope. Also, since they are transformed into parameters to the new loop-body-task, they are preserved as local to its scope, a scope that is self contained and unaltered from its original form. In that view, the transformation is totally consistent.

The aforementioned guaranties where tested exhaustively, using a number of programs. Below a few samples are presented; each was designed to test a specific property, thus all of them are relatively simple.
Figure 6.3: The fib program written in WOOL C

```c
wool int pfib( int n )
{
    if( n < 2 ) {
        return n;
    } else {
        int m,k;
        wool_spawn pfib (n-1);
        k = wool_call pfib (n-2);
        m = wool_sync pfib;
        return m+k;
    }
}

wool int main(int argc, char **argv )
{
    int n,m;
    if( argc < 2 ) {
        fprintf( stderr, "Usage: fib <woolopt>...<arg>\n" ),
        exit( 2 );
    }
    n = atoi( argv[ 1 ] );
    m = wool_call pfib(n);
    printf( "%d\n", m );
    return 0;
}
```

Figure 6.4: The fib program translated into GNU C. All external header files included have been removed for simplicity.

```c
TASK_1(int, pfib, int, n) {
    if (n < 2) {
        return n;
    } else {
        int m, k;
        SPAWN(pfib, n - 1);
        k = CALL(pfib, n - 2);
        m = SYNC(pfib);
        return m + k;
    }
}

TASK_2(int, main, int, argc, char **, argv) {
    int n, m;
    if (argc < 2) {
        fprintf(stderr, "Usage: fib <woolopt>...<arg>\n"), exit (2);
    }
    n = atoi(argv[1]);
    m = CALL(pfib, n);
    printf("%d\n", m);
    return 0;
}
```
```c
int main(int argc, char **argv )
{
    int grainsize = atoi( argv[1] );
    int p_iters = atoi( argv[2] );
    int s_iters = atoi( argv[3] );
    int i, f, g;
    for( i=0; i<s_iters; i++ ) {
        wool_for(f = 0; f <= p_iters; f++){
            int s = 0;
            for( g=0; g<grainsize; g++ ) {
                s += g;
            }
        }
    }
    printf( "%d %d %d\n", grainsize, p_iters, s_iters );
    return 0;
}
```

**Figure 6.5:** The loop program written in WOOL C

```c
LOOP_BODY_2(wool_for_0, LARGE_BODY, int, f, int, g, int, grainsize) { 
    int s = 0;
    for (g = 0; g < grainsize; g++) {
        s += g;
    }
}
```

```c
TASK_2(int, main, int, argc, char **, argv) {
    int grainsize = atoi(argv[1]);
    int p_iters = atoi(argv[2]);
    int s_iters = atoi(argv[3]);
    int i, f, g;
    for (i = 0; i < s_iters; i++) {
        FOR(wool_for_0, 0, p_iters, g, grainsize);
    }
    printf("%d %d %d\n", grainsize, p_iters, s_iters);
    return 0;
}
```

**Figure 6.6:** The loop program translated into GNU C. All external header files included have been removed for simplicity.
```c
#define MAX_ITEMS 256

struct item {
    int value;
    int weight;
};

int best_so_far = INT_MIN;

int compare(struct item *a, struct item *b) {
    double c = ((double) a->value / a->weight) -
                ((double) b->value / b->weight);
    if (c > 0)
        return -1;
    if (c < 0)
        return 1;
    return 0;
}

int read_input(const char *filename, struct item *items, int *capacity,
                int *n) {
    int i;
    FILE *f;
    if (filename == NULL)
        filename = "\0";
    f = fopen(filename, "r");
    if (f == NULL) {
        fprintf(stderr, "open_input("%s") failed\n", filename);
        return -1;
    }
    fscanf(f, "%d", n);
    fscanf(f, "%d", capacity);
    for (i = 0; i < *n; ++i)
        fscanf(f, "%d\%d", &items[i].value, &items[i].weight);
    fclose(f);
    qsort(items, *n, sizeof (struct item),
         (int (*)(const void *, const void *)) compare);
    return 0;
}
```

Figure 6.7: The knapsack program written in WOOL C, part I
wool int knapsack(struct item *e, int c, int n, int v) {
    int with, without, best;
    double ub;
    if (c < 0)
        return INT_MIN;
    if (n == 0 || c == 0)
        return v; /* feasible solution, with value v */
    ub = (double) v + c * e->value / e->weight;
    if (ub < best_so_far) { /* prune */
        return INT_MIN;
    }
    return knapsack(e + 1, c, n - 1, v);
    with = call knapsack(e + 1, c - e->weight, n - 1, v + e->value);
    without = sync knapsack;
    best = with > without ? with : without;
    if (best > best_so_far)
        best_so_far = best;
    return best;
}

char *specifiers[] = {"-f", "-benchmark", "-h", 0};
int opt_types[] = {STRINGARG, BENCHMARK, BOOLARG, 0};

wool int main(int argc, char **argv) {
    struct item items[MAX_ITEMS]; /* array of items */
    int n, capacity, sol;
    char filename[100];
    /* standard benchmark options */
    strcpy(filename, "knapsack-example2.input");
    get_options(argc, argv, specifiers, opt_types, filename);
    if (read_input(filename, items, &capacity, &n))
        return 1;
    sol = call knapsack(items, capacity, n, 0);
    printf("\nCilk Example: knapsack\n");
    printf("options: problem-file=%s\n", filename);
    printf("Best value is %d\n", sol);
    return 0;
}

Figure 6.8: The knapsack program written in WOOL C, part II
```c
struct item {
    int value;
    int weight;
};

int best_so_far = (-2147483647 - 1);

int compare(struct item *a, struct item *b) {
    double c = ((double) a->value / a->weight) - ((double) b->value / b->weight);
    if (c > 0) return -1;
    if (c < 0) return 1;
    return 0;
}

int read_input(const char *filename, struct item *items, int *capacity, int *n) {
    int i;
    FILE *f;
    if (filename == ((void *) 0)) filename = "\0";
    f = fopen(filename, "r");
    if (f == ((void *) 0)) {
        fprintf(stderr, "open_input("%s")_failed\n", filename);
        return -1;
    }
    fscanf(f, "%d", n);
    fscanf(f, "%d", capacity);
    for (i = 0; i < *n; ++i) fscanf(f, "%d", &items[i].value, &items[i].weight);
    fclose(f);
    qsort(items, *n, sizeof (struct item), (int (*)(const void *,
        const void *))compare);
    return 0;
}

int knapsack(struct item *e, int c, int n, int v) {
    int with, without, best;
    double ub;
    if (c < 0) return (-2147483647 - 1);
    if (n == 0 || c == 0) return v;
    ub = (double) v + c * e->value / e->weight;
    if (ub < best_so_far) {
        return (-2147483647 - 1);
    }
    SPAWN(knapsack, e + 1, c, n - 1, v);
    with = CALL(knapsack, e + 1, c - e->weight, n - 1, v + e->value);
    without = SYNC(knapsack);
    best = with > without ? with : without;
    if (best > best_so_far) best_so_far = best;
    return best;
}
```

Figure 6.9: The *knapsack* program translated into GNU C, part I. All external header files included have been removed for simplicity.
48

Evaluation

Figure 6.10: The knapsack program translated into GNU C, part II.

6.0.2 Performance issues

As mentioned in the introduction WOOL is rather similar to Cilk, especially syntactically. Since Cilk also comes with its own compiler, it came naturally to use it as reference for testing WCC. Both compilers where executed on the same machine, compiling a set of ten programs. These programs are part of the examples that are distributed with Cilk; this means that they are optimized for Cilk. Rewriting them in WOOL C consisted only of the most necessary syntactic changes, so in essence both version are equivalent. The version of Cilk used is 5.4.6. The C compiler and preprocessor used by WCC are those found in GCC version 4.4.4. The results are presented in the table below. The first two columns show the time needed to compile to binary by using CilkC and WCC respectively. The last column presents the time taken by WPP to translate the language to GNU C.

<table>
<thead>
<tr>
<th>program</th>
<th>CilkC (ms)</th>
<th>WCC (ms)</th>
<th>WPP (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bucket</td>
<td>255</td>
<td>416</td>
<td>147</td>
</tr>
<tr>
<td>cilksort</td>
<td>252</td>
<td>328</td>
<td>86</td>
</tr>
<tr>
<td>ck</td>
<td>264</td>
<td>320</td>
<td>8</td>
</tr>
<tr>
<td>fft</td>
<td>3.929</td>
<td>3.344</td>
<td>1.211</td>
</tr>
<tr>
<td>fib</td>
<td>175</td>
<td>199</td>
<td>5</td>
</tr>
<tr>
<td>heat</td>
<td>252</td>
<td>426</td>
<td>158</td>
</tr>
<tr>
<td>kalah</td>
<td>334</td>
<td>497</td>
<td>198</td>
</tr>
<tr>
<td>knapsack</td>
<td>179</td>
<td>294</td>
<td>62</td>
</tr>
<tr>
<td>lu</td>
<td>318</td>
<td>475</td>
<td>176</td>
</tr>
<tr>
<td>strassen</td>
<td>287</td>
<td>451</td>
<td>196</td>
</tr>
</tbody>
</table>

Table 6.1: Execution time comparison between CilkC, WCC and WPP. All times are given in milliseconds. Each result is the mean average of 10 runs.

For reference, here are some details about these programs. The first column is the lines of code (without any comments or blank lines); next is the number of tasks, of void tasks; finally is the set of arities produced.

The numbers only prove what was already mentioned earlier. The top-to-bottom approach of traversing the tree
Table 6.2: Details on the programs used for comparing CilkC and WCC. Lines of code counted with David A. Wheeler’s SLOCCount, version 2.26.

<table>
<thead>
<tr>
<th>program</th>
<th>LOC (Cilk)</th>
<th>LOC (WOOL)</th>
<th>TASKs</th>
<th>VOID_TASKs</th>
<th>Total tasks</th>
<th>arities</th>
</tr>
</thead>
<tbody>
<tr>
<td>bucket</td>
<td>200</td>
<td>172</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>cilksort</td>
<td>310</td>
<td>284</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>{2, 3, 5}</td>
</tr>
<tr>
<td>ck</td>
<td>391</td>
<td>367</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>{2}</td>
</tr>
<tr>
<td>fft</td>
<td>3092</td>
<td>3208</td>
<td>1</td>
<td>19</td>
<td>20</td>
<td>{0, 2, 3, 4, 5, 6, 8, 9}</td>
</tr>
<tr>
<td>fib</td>
<td>30</td>
<td>26</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>heat</td>
<td>290</td>
<td>240</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>{0, 2, 6}</td>
</tr>
<tr>
<td>knapsack</td>
<td>833</td>
<td>799</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>lu</td>
<td>123</td>
<td>101</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>{2, 4}</td>
</tr>
<tr>
<td>strassen</td>
<td>448</td>
<td>437</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>{7}</td>
</tr>
</tbody>
</table>

This table summarizes the details of the programs used for comparing CilkC and WCC. LOCs were counted with David A. Wheeler’s SLOCCount, version 2.26.

Gives a slight overhead in small and simple programs, while it is quite efficient in larger ones. FFT (fast Fourier transformation) spans well over three thousand lines of code, providing a high level of complexity. On all runs WCC proved to be faster than CilkC.

Further more, independently measuring the translation process gives one more insight. It is probably not WPP that is slow. Given that with most of the sample programs, the header file generated is more or less the same (similar collection of task arities), basic calculations can show that most of the excess time can safely be attributed to the Bash-based header file generation. An overall conclusion is that WPP and WCC have the potential of being refined further, making them both more robust and faster.

Figure 6.11: Plot: Execution time over lines of code. It is obvious how WCC scales nicely as the amount of code becomes larger.

Further more, independently measuring the translation process gives one more insight. It is probably not WPP that is slow. Given that with most of the sample programs, the header file generated is more or less the same (similar collection of task arities), basic calculations can show that most of the excess time can safely be attributed to the Bash-based header file generation. An overall conclusion is that WPP and WCC have the potential of being refined further, making them both more robust and faster.
**Figure 6.12:** Plot: Execution time over the total number of tasks. Having removed the FFT results, WCC scales much like CilkC; this proves that the performance of WCC is indeed related to the input and on an irrelevant bottleneck in the compiler itself.

**Figure 6.13:** Plot: Execution time over the maximum arity. Again without the FFT results, the arity shows the level of analysis required. Again the conclusion is the same that WCC scales much like CilkC, according to the input.
Conclusions

7.1 Conclusion

The overall project met the initial goals quite successfully. This is also true for my own goals since it was a journey that led to a wealth of new knowledge and experience. Having a fairly small previous background on low level programming languages like C, this proved early on to be a far from trivial project. There are two aspects to this. First of all, in order to write a scalable generic parser of the C language one has to get a strong grasp of the more inner and complicated parts of the language. Reaching this level from barely knowing the syntax was a huge undertaking that lasted for the duration. Also, a compiler is a complicated and multifaceted piece of software that has been highly researched and optimized in various ways. The traditional compiler construction theory does not even start to cover the true obstacles involved in a practical project. By building the compiler for C in C, it was crucial to master the language in order to actually apply it in the development also.

In higher level programming languages like Java, the most widely taught language in higher education today, there are plenty of software engineering aspects that are taken for granted; memory management, variable sized variables are just some examples. It is very possible for a programmer to go through a whole career without ever having to manually garbage collect variables. Having the opportunity to work on this project provided a great insight on how things actually work under the hood but also to truly comprehend a complicated programming language.

The most important lesson that I learned was to respect the theory and plan ahead. It happened more than once to have to discard whole files of code and start over just because there was this absolutely necessary feature not initially included; inflexible structures which might need to be extended for more features can become a true headache. However, exactly because a big part of the project was rewritten multiple times, the final version is a fairly decent implementation that would not have been done much differently on the possibility of redoing this project.

Final conclusion is that the development of a complicated language’s compiler is an exciting and fruitful experience that every software developer should take the time to do at least once. Apart from the aforementioned benefits, there is significant insight gained on how a programming language is transformed into machine code which is a great asset in the search for best practices. It is mainly the compiler that enforces specific programming methodologies to be better performing or more efficient than others.

7.2 Future work

There are a number of ways that this project can be extended in order to produce a more complete and better functioning compiler. The list below just names a few:

- Support more GNU Extensions.
- Handle macro *include* directives more intelligently.
- Perform full type analysis thus better enforcing specific constraints.
Conclusions

- Perform optimizations on specific code patterns.
- Extend code generation to actually produce C code rather than C macros.
- Implement a generic sync.
- The prototype issue.

7.2.1 GNU extensions

As said before, the parser in this project tried to support all the necessary GNU extensions that would allow it to parse the most common GNU C library header files, as well as general programming common practices. However, these extensions are far from a complete list. Most of the unsupported extensions have little to do with actual syntax or are indirectly supported. Some however, are simply not supported at all and in the case of having a source file that uses them the compiler will most probably abort on a Syntax Error.

The most notable amongst these unsupported extensions are:

- A big amount of more GCC built-in functions.
- Statements and Declarations in Expressions
- Locally Declared Labels
- Nested Functions

The built-in functions are easily handled with. One has to simply predefine them in the symboltable, exactly as done with the `__builtin_va_args` function, otherwise there would be parsing errors of symbols not being defined. So this comes down to simply compiling the list of these functions.

*Statements and declarations in expressions* is a rare GNU construct that is not supported by the official ISO C syntax since it allows compound statements to be used as if they were expressions. This is done with enclosing the compound statement in parenthesis, ending up with the following opening and closing pattern: `{ ... }`.

*Locally declared labels* use a new keyword `__label__` to declare labels in scopes other than the global. The current grammar would respond with a Syntactic Error upon encountering it, since the scanner will forward the `IDENTIFIER` token at its place. What needs to be done is scan the new token and create the grammatical rules for accepting such a statement inside compound statements.

Finally, *nested functions* do not seem simple to support. A function definition is not just a construct that defines a scope, it is an external declaration. However, to simply parse nested functions it is only necessary to allow a function definition to be used as a statement. It is not a trivial modification of the grammar but it is for sure much less trouble than actually trying to semantically understand this and transform it into machine code.

7.2.2 C macro Include directives

As explained in previous chapters, the mechanism introduced to avoid multiple inclusions of the same header is not at all up to the task, lacking support for important and common features of the C macro language. One way to fix this would be to create a new simple scanner that captures C macro conditional blocks; then if a include directive is defined in their body remove it, and only it, from the original file; finally, after preprocessing
reintroduce the conditional block with only the include directive in its body. This is not a real solution as it is an effective workaround to a problem that is of as major importance as it is not trivial to solve.

7.2.3 Type analysis

Currently the declarations table accepts the type of a declared symbol as nothing more than a string of characters. Parsing has guaranteed the syntactic validity of each one and the project requires not understanding of the types themselves. They are only printed out in different places. However, being able to actually make the compiler know the type would open a lot of new possibilities of optimizations. One example would be with the step-identifier of a parallel FOR loop which would allow for fairly complicated task splitting strategies to be automatically applied. Being able to know the identifiers type would allow for the compiler to understand how it changes between iterations and conclude in the most suitable way to split it into groups of iterations more efficiently.

7.2.4 Optimize specific patterns

Generic optimization is a parallelization framework is really hard to achieve if not impossible with the current understanding of its underlying mathematical concepts. However, there are specific occasions were various optimizations could be successfully applied. It would be an interesting sub-project to allow the back-end to capture such patterns and transform them accordingly.

Such a scenario would be to automatically rearrange the instructions of tasks into a different set of tasks that can be much more efficient. For example if task A spawns B which then spawns C, there are configurations where it would prove much more efficient to merge B and C or A with B or simply splitting B into both A and C. These transformation would of course be done with the most internal operations of WOOL in mind, trying to minimize overhead and maximize parallelization. These scenarios are very specific and not meant for general practice, although not at all uncommon since many times developers forget of the overhead that comes with spawning and computing new tasks.

7.2.5 Output C code rather than C macros

This could prove to be a double edged knife in the sense that making the compiler able to manipulate actual C code would be greatly beneficial to the code generation aspects of the compiler. If that stage is reached it wouldn’t be far since the compiler would be able to adapt certain aspects of the WOOL library to the exact requirements of each project.

Of course reaching the level where the compiler is able to generate custom C code is far from trivial and it would take considerable amounts of time. Moreover, during the initial stages of this process it would be impossible to modify the WOOL library in any significant way as it would may break the compiler. This is a significant drawback to be enforced on a project that is under active development, so this project shouldn’t be considered before WOOL’s development has reached production level.
7.2.6 A generic sync

A generic sync is nothing more than a combination of the aforementioned projects, or better yet an application of some of them. It concerns the implementation of a SYNC mechanism that accepts no argument and automatically syncs all tasks up to its scope. It might sound easy but it has many pitfalls. There are two ways to implement this... in run-time and during compile time. Since the first case is obviously outside the boundaries of this project, let's take a look at the second.

In order to implement a generic sync during compile time, the compiler has to substitute it with the appropriate normal SYNC operations. To do that it requires to know the names of the tasks to be synced and their order. However, during compile time there is no stack to traverse and find out, so it is necessary to have the compiler traverse the source code and manually find out all that it needs. This task is far more complicated than it sounds since there are many cases where it might even be impossible. Imagine the common case of a while loop that spawns a task according to an if statement that changes according to some iteration counter. Or worse yet a task calling another task that calls an other task and all of them spawning more tasks.

One more obstacle is the problem of the return value. Since a single generic sync corresponds to multiple spawns, where is their return value returned to? The solution to this is as easy as having a return variable pointer be passed along to the SPAWN operation.

Nevertheless, it is easy to understand how the complexity can exponentially become simply impossible. However, there are specific patterns where such a scenario would become feasible or methodologies to follow that would ensure fault tolerance. However, the run-time solution is so much more efficient that it would render most of them unnecessary.

7.2.7 The prototype issue

This last part is actually an important shortcoming of WOOL as a library and inherited by WCC also. When defining a task, the macros are expanded in place of the definition which include a number of real functions. It is calls of these same functions that the Call, Spawn and Sync macros expand to. Also, the syntax does not offer a way of including task prototypes which would expand to the appropriate forward declarations. This issue becomes a major problem when there are mutually recursive tasks in the program and will always result in a "symbol not defined" error.

The only way to overcome this issue is by extending the syntax into including prototypes for tasks. Actually work on this is already under way and it is to be completed really soon. The actual semantics of the modification span throughout the project. Starting from the parser which has to allow a body-less task definition (for the actual syntactic rule see figure 7.1), moving on to the AST which has to allow this as both a new type of external declaration but also analyse it and extract all the pieces of information required. Analysis is needed because there are three forward declarations to be generated and in contrast to the rest of the project these are GNU C code directly. The reason for this inconsistency is that the WOOL project itself does not yet support this feature in its released versions. Finally, this modification affects almost any other part of the compiler, including the two tables, since a forward declaration is nothing more than a symbol declaration which has to be recognised before the actual full definition of the task.
Figure 7.1: A new WOOL task syntax to solve the prototype issue. The old syntax would not work since it wouldn’t allow for any new alternatives. In reality the production of the function definition is used directly, with the prototype as a first alternative production.

```c
wool int main(int argc, char **argv);

void SPAWN_main(Task *__dq_top, int argc, char **argv);

int CALL_main(Task *__dq_top, int argc, char **argv);

int SYNC_main(Task *__dq_top);
```

Figure 7.2: A sample task prototype translation. On top there is a typical wool task given as a prototype. Below is the expansion of it, directly into GNU C. These three functions correspond to the functions being called for the CALL, SPAWN and SYNC operation respectively. For a detailed explanation of these functions, the reader is redirected to the WOOL documentation.
The final vocabulary

The scanner was built using GNU Flex. Thus the vocabulary is written using the GNU Flex specification language [19] [20]. In short the vocabulary is divided by multiple rule entries, each rule taking up one line. The rule itself can be divided into two parts. On the left there is a regular expression pattern that is used to match certain groups of consecutive characters. On the right and inside braces there are the action rules which correspond to instructions that are executed upon successful match of the corresponding pattern. For the purposes of our project these action rules do nothing more than assign and return a token for each of the rules. It is important to mention that the rules are matched on a first come first served basis, starting from the top of the file. So, the order of the rules is of the utmost importance. Finally, some the original action codes are more complex; the extra code has been striped out as it is of no importance to understanding the vocabulary.

```flex
/* some complex named patterns */
/* Universal Character Name */
UCN (\\u[0-9a-fA-F]{4}|\\U[0-9a-fA-F]{8})
/* float exponent */
EXP ([Ee][-+]?[0-9]+)
/* integer length */
ILEN ([Uu](L|l|LL|ll)?|(L|l|LL|ll)[Uu]?)

%option noyywrap
%option yylineno
%x SC_COMMENT
%x SC_ASM
%

/* match the next line and keep it buffered for error reporting purposes */


/* free the previous line */
if(linebuf != NULL) free(linebuf);
/* save the next line */
linebuf = strdup(yytext);
if(DEBUG_PARSE)
  /* makes easier to map tokens to source */
  fprintf(stderr, "\n %4d: ",yylineno);
/* give back all but the <NEW_LINE> to rescan */
```
yyless(1);

/* match comments */

"/*" { BEGIN(SC_COMMENT); }
<SC_COMMENT>"*/" { BEGIN(previous_sc); }
<SC_COMMENT>([^*]|
)+.|<SC_COMMENT><<EOF>> { yyerror("error: Unterminated comment");
  abort(); }
"//".*\n
/* All things WOOL related are in this section */

(wool_for|pfor) { return token(_W_FOR); }
(wool_spawn|spawn) { return token(_W_SPAWN); }
(wool_call|call) { return token(_W_CALL); }
(sync) { return token(_W_SYNC); }
(wool|task) { return token(_W_TASK); }

/* GCC extensions */

__extension__ { return token(EXTENSION); }
__attribute__[ \t]*"((\.*))" { return token(ATRIBUTE); }

asm|__asm__([ \t]+(volatile|goto))?[ \t]*\([ \t]*.*\) { return token(ASM); }

__const { return token(CONST);}
__const__ { return token(_CONST_);}
__restrict { return token(_RESTRICT);}
__restrict__ { return token(_RESTRICT_);}

/* ISO C vocabulary from here on */

/* declaration keywords */

_Bool { return token(BOOL);}
_Complex { return token(COMPLEX);}
_Imaginary { return token(IMAGINARY);}
char { return token(CHAR);}
double { return token(DOUBLE); }
float { return token(FLOAT); }
int { return token(INT); }
long { return token(LONG); }
short { return token(SHORT); }
signed { return token(SIGNED); }
unsigned { return token(UNSIGNED); }
void { return token(VOID); }
auto { return token(AUTO); }
const { return token(CONST); }
extern { return token(EXTERN); }
inline { return token(INLINE); }
static { return token(STATIC); }
register { return token(REGISTER); }
restrict { return token(REstrict); }
volatile { return token(VOLATILE); }
enum { return token(ENUM); }
struct { return token(STRUCT); }
union { return token(UNION); }
typedef { return token(TYPedef); }

/* keywords */
break { return token(BREAK); }
case { return token(CASE); }
continue { return token(ELLIPSIS); }
default { return token(DEFAULT); }
goto { return token(GOTO); }
return { return token(REturn); }
sizeof { return token(SIZEOF); }
switch { return token(SWITCH); }
do { return token(DO); }
else { return token(ELSE); }
for { return token(FOR); }
if { return token(IF); }
while { return token(WHILE); }

/* integers */
0[0-7]*{ILEN}? { return setconstval(CONSTANT_INT); }
[1-9][0-9]*{ILEN}? { return setconstval(CONSTANT_INT); }
0[Xx][0-9a-fA-F]+{ILEN}? { return setconstval(CONSTANT_INT); }
/* decimal float */

([0-9]*\.[0-9]+|[0-9]+\.)(EXP)?(f|F|L|l)?
  { return setconstval(CONSTANT_FLO); }

[0-9]+(EXP)f(l|L)?
  { return setconstval(CONSTANT_FLO); }

/* hex float */

0[Xx]([0-9a-fA-F]*\.[0-9a-fA-F]+|[0-9a-fA-F]+\.?)?Pp[-+]?[0-9]+(f|F|L|l)?
  { return setconstval(CONSTANT_STR); }

/* char const */

'([^'\\]|\\['"\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9a-fA-F]+|UCN)+'
  { return setconstval(CONSTANT_STR); }

/* string literal */

L?"([^"\\]|\\['"\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9a-fA-F]+|UCN)+"
  { return setconstval(CONSTANT_STR); }

/* punctuators */

"..."
  { return token(ELLIPSIS); }

">="
  { return token(RIGHT_ASSIGN); }

"<=="
  { return token(LEFT_ASSIGN); }

"+="
  { return token(ADD_ASSIGN); }

"-="
  { return token(SUB_ASSIGN); }

"*="
  { return token(MUL_ASSIGN); }

="/="
  { return token(DIV_ASSIGN); }

"%="
  { return token(MOD_ASSIGN); }

"&="
  { return token(AND_ASSIGN); }

"^="
  { return token(XOR_ASSIGN); }

"|="
  { return token(OR_ASSIGN); }

">>=
  { return token(RIGHT_OP); }

"<<=
  { return token(LEFT_OP); }

"++=
  { return token(INC_OP); }

"--=
  { return token(DEC_OP); }

"->=
  { return token(PTR_OP); }

"&&=
  { return token(AND_OP); }
The final vocabulary

"||" { return token(OR_OP); }
"<=" { return token(LE_OP); }
">=" { return token(GE_OP); }
"==" { return token(EQ_OP); }
"!=" { return token(NE_OP); }
";" { return token(';'); }
"{"|"<%" { return token('{'); }
"}"|"%>" { return token('}'); }
"," { return token(','); }
":" { return token(':'); }
"=" { return token('='); }
"(" { return token('('); }
")" { return token(')'); }
"["|"<:" { return token('['); }
"]"|":>" { return token(']'); }
")." { return token('.'); }
"&" { return token('&'); }
"!" { return token('!'); }
"~" { return token('~'); }
"-" { return token('-'); }
"+" { return token('+'); }
"*" { return token('*'); }
"/" { return token('/'); }
"\" { return token('%'); }
"<" { return token('<'); }
">" { return token('>'); }
"^" { return token('^'); }
"?" { return token('?'); }

/* new identifier */

([^a-zA-Z]|{UCN})([_a-zA-Z0-9]|{UCN})* { return ret_symbol(); }

/* whitespace discarded */

[ \t]+

/* line continuations also discarded */

\\$
/* some preprocessor stuff that are simply discarded */

[]*""[]*include[]*+[]*([[]*""[]*\*\*)|([]*""[]*\*\*)

[]*""[]*(*if|else|endif|define|undef|line|pragma).*

{unmatched("error: found unhandled preprocessor directive");}

/* Unmatched characters */

. {unmatched("error: found unmatched character");}

%%
The final grammar

The parser was built using GNU Bison. Thus the grammar is written using the GNU Bison specification language [19] [8]. In short the grammar is divided by multiple rule entries, each rule ending with a semi colon. The rule itself can be divided into two parts. First comes the non-terminal sentence to be reduced from the combination of other sentences. Next and after the colon comes that combination which can consist of any terminal or non-terminal sentence or token. Any rule can have multiple alternative combinations to be reduced from. These alternatives are delimited by a pipe character.

Wherever in the grammar there were special modifications or additions made there are explanatory comments. If no comment states otherwise the rule was taken as is from the ISO C specification.
%token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
%token _RESTRICT _RESTRICT_ _CONST _CONST_
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token BOOL COMPLEX IMAGINARY
%token STRUCT UNION ENUM ELLIPSIS
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
%token ATTRIBUTE ASM EXTENSION
%token _W_FOR _W_SPAWN _W_CALL _W_SYNC _W_TASK
%type <intval> CONSTANT_INT
%type <floval> CONSTANT_FLO
%type <strval> CONSTANT_STR
%type <strval> IDENTIFIER
%type <strval> STRING_LITERAL
%type <strval> TYPE_NAME
%type <strval> ATTRIBUTE
%type <strval> ASM asm_expression
%type <intval> EXTENSION
%type <ast> attribute_specifier attribute_specifier_list asm_statement
%type <ast> struct_declaration_extended declaration_extended
%type <intval> _W_FOR _W_SPAWN _W_CALL _W_SYNC _W_TASK
wool_task_specifier
%type <ast> wool_task_definition
%type <ast> wool_for_init_statement
%type <ast> wool_for_test_statement
%type <intval> wool_for_test_op
%type <ast> wool_for_step_expression
%type <ast> wool_expression wool_call_expression wool_sync_expression
%type <ast> wool_for_statement wool_spawn_statement
%type <intval> SIZEOF
%type <intval> PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%type <intval> AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%type <intval> SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
The final grammar

%type <intval> XOR_ASSIGN OR_ASSIGN
%type <intval> ';' '{' '}' ',' ':' '=' '(' ')' '[' ']' '.' '&' '!' 
%type <intval> '~' '-' '+' '*' '/' '%' '<' '>' '^' '|' '?'
%type <intval> TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
%type <intval> CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE
%type <intval> CONST VOLATILE VOID
%type <intval> _RESTRICT _RESTRICT_ _CONST _CONST_
%type <intval> BOOL COMPLEX IMAGINARY
%type <intval> STRUCT UNION ENUM ELLIPSIS
%type <intval> GOTO CONTINUE BREAK RETURN
%type <intval> assignment_operator unary_operator struct_or_union

%type <ast> pointer
%type <ast> init_declarator init_declarator_list
%type <ast> initializer initializer_list
%type <ast> designator designator_list designation
%type <ast> identifier_list direct_declarator declarator
declarator_attributed
%type <ast> direct_abstract_declarator abstract_declarator
%type <ast> declaration_specifiers specifier_qualifier_list
%type <ast> storage_classSpecifier typeQualifier typeQualifier_list
%type <ast> typeSpecifier functionSpecifier
%type <ast> statement labeled_statement compound_statement
expression_statement
%type <ast> if_statement selection_statement iteration_statement
jump_statement
%type <ast> block_item block_item_list
%type <ast> expression unary_expression constant_expression
cast_expression
%type <ast> multiplicative_expression additive_expression
shift_expression
%type <ast> relational_expression equality_expression and_expression
%type <ast> exclusive_or_expression inclusive_or_expression
%type <ast> logical_and_expression logical_or_expression
%type <ast> conditional_expression assignment_expression
%type <ast> primary_expression postfix_expression
%type <ast> argument_expression_list
%type <ast> type_name
%type <ast> struct_or_union_specifier struct_declarator
struct_declarator_list
%type <ast> struct_declarator struct_declarator_list
%type <ast> declaration parameter_declaration parameter_list
parameter_type_list
%type <ast> declaration_list
%type <ast> enum_specifier enumerator enumerator_list


%type <ast> function_definition external_declaration
%type <ast_root> translation_unit

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE DUMMY
%nonassoc ','

%start translation_unit
%

primary_expression
: IDENTIFIER
 | CONSTANT_INT
 | CONSTANT_FLO
 | CONSTANT_STR
 | STRING_LITERAL
 | '(' expression ')' ;

postfix_expression
: primary_expression
 | postfix_expression '[' expression ']'  
 | postfix_expression '(' ')' 
 | postfix_expression '('. argument_expression_list ')'
 | postfix_expression '.' IDENTIFIER
 | postfix_expression PTR_OP IDENTIFIER
 | postfix_expression INC_OP
 | postfix_expression DEC_OP
 | '(' type_name ')' '{' initializer_list '}'
 | '(' type_name ')' '{' initializer_list ',' '}' ;

argument_expression_list
: assignment_expression
 | argument_expression_list ',' assignment_expression
 |

unary_expression
: postfix_expression
 | INC_OP unary_expression
 | DEC_OP unary_expression
 | unary_operator cast_expression
| SIZEOF unary_expression
| SIZEOF '() type_name ')' |

;

unary_operator
|
| '+'
| '*'
| '+'
| '-'
| '-'
| '!' |

;

cast_expression
:
 unary_expression
| '() type_name ')' cast_expression
|

;

multiplicative_expression
:
 cast_expression
| multiplicative_expression '*' cast_expression
| multiplicative_expression '/' cast_expression
| multiplicative_expression '%' cast_expression
|

;

additive_expression
:
 multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
|

;

shift_expression
:
 additive_expression
| shift_expression LEFT_OP additive_expression
| shift_expression RIGHT_OP additive_expression
|

;

relational_expression
:
 shift_expression
| relational_expression '<' shift_expression
| relational_expression '>' shift_expression
| relational_expression LE_OP shift_expression
| relational_expression GE_OP shift_expression
|

;
equality_expression
  : relational_expression
  | equality_expression EQ_OP relational_expression
  | equality_expression NE_OP relational_expression
  
and_expression
  : equality_expression
  | and_expression '&&' equality_expression
  
exclusive_or_expression
  : and_expression
  | exclusive_or_expression '^^' and_expression
  
inclusive_or_expression
  : exclusive_or_expression
  | inclusive_or_expression '|' exclusive_or_expression
  
logical_and_expression
  : inclusive_or_expression
  | logical_and_expression AND_OP inclusive_or_expression
  
logical_or_expression
  : logical_and_expression
  | logical_or_expression OR_OP logical_and_expression
  
conditional_expression
  : logical_or_expression
  | logical_or_expression '?' expression ':' conditional_expression
  
assignment_expression
  : conditional_expression
  | wool_expression
  | unary_expression assignment_operator assignment_expression
  
assignment_operator
  : '='
  | MUL_ASSIGN
expression : assignment_expression
            | expression ',' assignment_expression
            ;

constant_expression : conditional_expression
                    ;

declaration_extended : EXTENSION declaration
                      | declaration
                      ;

declaration : declaration_specifiers init_declarator_list ';'
              | declaration_specifiers ';'
              ;

declaration_specifiers : storage_class_specifier
                        | storage_class_specifier declaration_specifiers
                        | type_specifier declaration_specifiers
                        | type_specifier
                        | type_qualifier declaration_specifiers
                        | type_qualifier
                        | function_specifier
                        | function_specifier declaration_specifiers
                        ;

init_declarator_list : init_declarator
                     | init_declarator_list ',' init_declarator
                     ;
```
init_declarator
    : declarator_attributed
    | declarator_attributed '=' initializer
    ;

declarator_attributed
    : declarator
    | attribute_specifier_list declarator
    | declarator attribute_specifier_list
    ;

storage_classSpecifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
    ;

typeSpecifier
    : VOID
    | CHAR
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | BOOL
    | COMPLEX
    | IMAGINARY
    | struct_or_unionSpecifier
    | enumSpecifier
    | TYPE_NAME
    ;

struct_or_unionSpecifier
    : struct_or_union IDENTIFIER '{' struct_enter struct_declaration_list struct_leave '}'
    | struct_or_union attributeSpecifier_list IDENTIFIER '{' struct_enter struct_declaration_list struct_leave '}'
    | struct_or_union '{' struct_enter struct_declaration_list struct_leave '}'
```
The final grammar

struct_or_union attribute_specifier_list '{' struct_enter
    struct_declaration_list struct_leave '}'
| struct_or_union IDENTIFIER
|
;

struct_or_union
 : STRUCT
 | UNION
 |
;

struct_declaration_list
 : struct_declaration_extended
 | struct_declaration_list struct_declaration_extended
 |
;

struct_declaration_extended
 : EXTENSION struct_declaration
 | struct_declaration
 |
;

struct_declaration
 : specifier_qualifier_list struct_declarator_list ';'
 | struct_or_union_specifier ';'
 |
;

specifier_qualifier_list
 : typeSpecifier specifier_qualifier_list
 | typeSpecifier
 | type_qualifier specifier_qualifier_list
 | type_qualifier
 |
;

struct_declarator_list
 : struct_declarator
 | struct_declarator_list ',' struct_declarator
 |
;

struct_declarator
 : declarator
 | '::' constant_expression
 | declarator '::' constant_expression
 |
;

eNumspecfier
 : ENUM '{' struct_enter enumerator_list struct_leave '}'}
| ENUM IDENTIFIER '{' struct_enter enumerator_list struct_leave '}' |
| ENUM '{' struct_enter enumerator_list ',' struct_leave '}' |
| ENUM IDENTIFIER '{' struct_enter enumerator_list ',' struct_leave '}' |
| ENUM IDENTIFIER |
| ENUM IDENTIFIER |

enumerator_list
: enumerator
| enumerator_list ' ' , ' , enumerator |
| enumerator |

enumerator
: IDENTIFIER |
| IDENTIFIER ' ' = ' constant_expression |
| IDENTIFIER ' ' = ' constant_expression |
| IDENTIFIER |

function_specifier
: INLINE |
| INLINE |

declarator
: pointer direct_declarator asm_expression |
| pointer direct_declarator |
| direct_declarator asm_expression |
| direct_declarator |

| direct_declarator |
| ' ( ' declarator ' ) ' |
| direct_declarator '[' ' type_qualifier_list assignment_expression ' ] ' |
| direct_declarator '[' ' type_qualifier_list ' ] ' |
| direct_declarator '[' ' assignment_expression ' ] ' |
direct_declarator [' STATIC type_qualifier_list assignment_expression ']
| direct_declarator [' type_qualifier_list STATIC assignment_expression ']
| direct_declarator [' type_qualifier_list '*' ']
| direct_declarator [' '*' ']
| direct_declarator [' ]'
| direct_declarator (' funcpar_enter parameter_type_list ')
| direct_declarator (' identifier_list ')
| direct_declarator (' ')
;

pointer
: ' *'
| '*' type_qualifier_list
| '*' pointer
| '*' type_qualifier_list pointer
;

type_qualifier_list
: type_qualifier
| type_qualifier_list type_qualifier
;

parameter_type_list
: parameter_list
| parameter_list ',' ELLIPSIS
;

parameter_list
: parameter_declaration
| parameter_list ',' parameter_declaration
;

parameter_declaration
: declaration_specifiers declarator
| declaration_specifiers abstract_declarator
| declaration_specifiers
;

identifier_list
: IDENTIFIER
| identifier_list ',' IDENTIFIER
;
type_name
  : specifier_qualifier_list
  | specifier_qualifier_list abstract_declarator
  ;

abstract_declarator
  : pointer
  | direct_abstract_declarator
  | pointer direct_abstract_declarator
  ;

direct_abstract_declarator
  : '(' abstract_declarator ')
  | '[' ']' 
  | '[' assignment_expression ']
  | direct_abstract_declarator '[' ']' 
  | direct_abstract_declarator '[' assignment_expression ']
  | '[' '*' ']
  | direct_abstract_declarator '[' '*' ']
  | '(' ')
  | '(' scope_enter parameter_type_list scope_leave ')
  | direct_abstract_declarator '('
  | direct_abstract_declarator '('
  | direct_abstract_declarator '('
  | direct_abstract_declarator '('
  | direct_abstract_declarator '('

initializer
  : assignment_expression
  | '{' initializer_list '}
  | '{' initializer_list ',' '}
  ;

initializer_list
  : initializer
  | designation initializer
  | initializer_list ',' initializer
  | initializer_list ',' designation initializer
  ;

designation
  : designator_list '=
  ;

designator_list
The final grammar:

```
: designator  
| designator_list designator  
;  

designator  
: '[' constant_expression ']'  
| '.' IDENTIFIER  
;  

statement  
: labeled_statement  
| scope_enter compound_statement scope_leave  
| expression_statement  
| selection_statement  
| iteration_statement  
| jump_statement  
| asm_statement  
| wool_for_statement  
| wool_spawn_statement  
;  

labeled_statement  
: IDENTIFIER ':' statement  
| IDENTIFIER ':' attribute_specifier_list statement  
| CASE constant_expression ':' statement  
| DEFAULT ':' statement  
;  

compound_statement  
: '{' '}'
| '{' block_item_list '}'
;  

block_item_list  
: block_item
| block_item_list block_item  
;  

block_item  
: declaration_extended
| statement  
;  

expression_statement  
: ';'
```
if_statement
  : IF '(' expression ')' statement %prec LOWER_THAN_ELSE
  | IF '(' expression ')' statement ELSE statement
  ;

selection_statement
  : if_statement
  | SWITCH '(' expression ')' statement
  ;

iteration_statement
  : WHILE '(' expression ')' statement
  | DO statement WHILE '(' expression ')' ';
  | FOR '(' expression_statement expression_statement ')' statement
  | FOR '(' expression_statement expression_statement expression ')' statement
  | FOR '(' declaration expression_statement expression ')' statement
  | FOR '(' declaration expression_statement expression ')' statement
  ;

jump_statement
  : GOTO IDENTIFIER ';
  | CONTINUE ';
  | BREAK ';
  | RETURN ';
  | RETURN expression ';
  ;

translation_unit
  : external_declaration
  | translation_unit external_declaration
  ;

external_declaration
  : function_definition
  | wool_task_definition
  | declaration_extended
  ;

function_definition
  : declaration_specifiers declarator declaration_list function_enter
    compound_statement function_leave
The final grammar

| declaration_specifiers declarator function_enter compound_statement function_leave |
| declaration_list |
| attributeSpecifier |
| attributeSpecifierList |
| asmExpression |
| asmStatement |
| woolTaskSpecifier |
| woolTaskDefinition |
| woolForStatement |
| woolForInitStatement |
| specifierList declarator '=' initializer ';;' |
;  

wool_for_test_statement
: IDENTIFIER wool_for_test_op expression ';';
;

wool_for_test_op
: '<'
| '>'
| LE_OP
| GE_OP
;

wool_for_step_expression
: INC_OP IDENTIFIER
| DEC_OP IDENTIFIER
| IDENTIFIER INC_OP
| IDENTIFIER DEC_OP
| IDENTIFIER ADD_ASSIGN expression
| IDENTIFIER SUB_ASSIGN expression
;

wool_expression
: wool_call_expression
| wool_sync_expression
;

wool_spawn_statement
: _W_SPAWN IDENTIFIER '(' expression ')';
;

wool_call_expression
: _W_CALL IDENTIFIER '(' expression ')'
;

wool_sync_expression
: _W_SYNC IDENTIFIER
;

funcpar_enter
: scope_enter
;
function_enter
  : scope_enter
  ;

function_leave
  : scope_leave
  ;

struct_enter
  : scope_enter
  ;

struct_leave
  : scope_leave
  ;

parlist_enter
  : scope_enter
  ;

parlist_leave
  : scope_leave
  ;

scope_enter
  :
  ;

scope_leave
  :
  ;

%%
The Abstract Syntax Tree

1 _AST_seq
2 : argument_expression_list
3 | declaration_specifiers
4 | init_declarator_list
5 | declaration_list
6 | enumerator_list
7 | declaration_specifiers
8 | designator_list
9 | struct_declaration_list
10 | specifier_qualifier_list
11 | type_qualifier_list
12 | parameter_type_list
13 | parameter_list
14 | identifier_list
15 | block_item_list
16 | attribute_specifier_list
17 ;

19 _AST_terminal
20 : IDENTIFIER
21 | CONSTANT_INT
22 | CONSTANT_FLO
23 | CONSTANT_STR
24 | STRING_LITERAL
25 | TYPEDEF
26 | EXTERN
27 | STATIC
28 | AUTO
29 | REGISTER
30 | VOID
31 | CHAR
32 | SHORT
33 | INT
34 | LONG
35 | FLOAT
36 | DOUBLE
37 | SIGNED
38 | UNSIGNED
39 | BOOL
40 | COMPLEX
The Abstract Syntax Tree

| IMAGINARY |
| TYPE_NAME |
| CONST |
| _CONST |
| _CONST_ |
| RESTRICT |
| _RESTRICT |
| _RESTRICT_ |
| VOLATILE |
| INLINE |
| ASM |

; 

_AST_declaration

: EXTENSION* declaration_specifiers init_declarator_list* ';'
| EXTENSION* specifier_qualifier_list struct_declarator_list ';
| EXTENSION* struct_or_unionSpecifier ';
| ;

_AST_declarator

: IDENTIFIER =* exp*
| declarator = declarator
| declarator* : exp*
| attr_list* declarator attr_list*
| ( declarator )
| declarator [ STATIC* type_qualifier_list* exp* ]
| declarator [ type_qualifier_list* STATIC* exp* ]
| declarator [ * ]
| declarator ( (parameter_list|id_list)* )
| pointer declarator*
| declarator* [ exp* ]
| declarator* [ * ]
| declarator* ( parameter_type_list* )
| { declarator_list ,* }
| [ exp ]
| . id
| specifier_qualifier_list declarator = declarator ;
| ;

_AST_type_name

: specifier_qualifier_list declarator*

;

_AST_pointer

: * type_qualifier_list* pointer*
; 

_AST_struct
: STRUCT attr_list* id* { struct_declaration_list }
| STRUCT id
| UNION attr_list* id* { struct_declaration_list }
| UNION id
| ENUM id* { enumerator_list,* }
;

_AST_function
: (wool|task)* declaration_specifiers declarator declaration_list* stmt
;

_AST_attribute
: ATTRIBUTE ,*
;

_AST_exp
: id
| const
| exp op exp*
| op exp
| exp ( )
| exp ( list )
| exp [ exp ]
| SIZEOF ( type )
| { type }{ list }
| { type } exp
| exp , exp
;

_AST_conditional
: exp ? exp : exp
| IF ( exp ) stmt ELSE* stmt*
;

_AST_flow
: id : attr_list* stmt
| CASE exp : stmt
| DEFAULT : stmt
| exp* ;
<table>
<thead>
<tr>
<th>SWITCH ( exp ) stmt</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE ( exp ) stmt</td>
</tr>
<tr>
<td>DO stmt WHILE ( exp ) ;</td>
</tr>
<tr>
<td>GOTO id ;</td>
</tr>
<tr>
<td>(CONTINUE</td>
</tr>
<tr>
<td>RETURN exp* ;</td>
</tr>
<tr>
<td>;</td>
</tr>
</tbody>
</table>

_AST_loop

: FOR ( exp* exp* exp* ) stmt |
| FOR ( decl exp* exp* ) stmt |
| _W_FOR ( wool_for_init_statement wool_for_test_statement wool_for_step_expression ) stmt |
| ; |
The TAB-table structure

The TAB-table structure, or better yet type, is a complicated linked-list based mechanism that is used throughout the project offering invaluable support. The basic idea behind it is two offer organization of data on two layers. This could resemble a typical two dimensional array, although it is a bit more complicated than that.

The use cases of the table in this project enforce a single constraint over its design. This is that the second dimension must have variable size throughout its scope which theoretically can become large enough to be considered infinite. This assumption helps keep in par with the worst case scenario although it is quite impossible to happen\(^1\). However, complicating things to meet the ideal model for the requirements actually gave birth to a number of further advantages which are altogether presented as properties in the following list.

1. Allow for variable-sized addressing of the second dimension.
2. Allow to map several records to the same key in reversed chronological order.
3. Direct access to the latest record.
4. Fast lookup of the latest record of an entry.

The actual structure of the table is as follows... The second dimension is implemented as a typical stack allowing for multiple entries of the same record to be assigned to the same key in a reversed chronological order, as well as generally having variable-sized storage capacity. Each record in each stack is called a Tab-cell; each stack as a whole is called a row. The first dimension is a fixed-length array, with its boundary set at 9997 entries.

Entries are stored given a single key; this key is then hashed modulo 9997. This procedure provides three guarantees:

1. An alphanumeric key is converted to a positive numerical value.
2. Numerical keys less than 9997 remain the same.
3. There is no possibility of an overflow.

The benefits of the above architecture are plenty, all of course in combination to the specific requirements of this project. The first guarantee is what makes property 4 possible. Iterating a specific row will lead to the latest record of the entry in question relatively fast. The second guarantee makes it possible for the TAB-table type to resemble a typical one dimensional array which always returns the latest record of an entry. The key aspect is that by using numerical keys, one knows exactly which records end up in which rows. Guarantee three’s benefits are obvious.

The wrapper TAB-table structure is comprised of two fields. The fixed-size array of rows and the top variable holding the index of the row that was added a record last. Since, all rows are stacks, the head of the top-row is also the latest cell to be added to the table.

A cell holds a void pointer to its actual contents, a pointer to the next cell (which in context is actually the previous record, since new records are prepended) and a backup reference of the index of the row that was top.

---

\(^1\)Assuming the limit of a 16bit integer at 32768, this means 32768 symbols with the same hash value; also, taking into account the usage of such tables this number does not represent a complete source file but only a single scope nesting path, since upon exiting a scope all entries related to it are deleted. With all that in mind it is safe to assume that 32768 is an amount of custom defined symbols which cannot be easily reached.
before this cell was added. When removing a cell its top reference will be restored as the table’s top variable value thus resuming its state as if the cell was never added.

![Graphical representation of a sample TAB-table.](image)

**Figure D.1:** Graphical representation of a sample TAB-table. Top has a reference to the row with the latest cell. Each cell has a reference to the row that had the latest cell before itself.

A TAB-cell has been designed with a void pointer as the container for its contents. This is very helpful for reusing the TAB-table type in multiple situation where different data has to be stored and organized in this manner. However, allowing for application level code to use void pointers is like asking for trouble since it can generate a big amount of errors and bugs that many times are hard to pinpoint there origin. To that extend, except using the predefined API for every interaction with the table, an API that is fairly elaborate in what types of interaction it allows, it was seen as best practice to create wrapper sets of functions for every specific type that it is going to be stored to an instance of the table, minimizing the use of void pointers everywhere in the application level of the project.
The stack structure

One more structure used as a common type in this project is stack. This follows the architecture of a common last-in-first-out (LIFO) linked-list. Let’s see it’s structure in more detail... This simple structure is composed of a void pointer fixed-size array and an integer variable that works as a reference to the array’s index that holds the latest value. Since this structure is LIFO, additions always happen at the top+1 index; removing an item is only possible for the latest one (index top); that way mass removal of item must happen sequentially. There is no random access available. However, there are two functions for accessing an element; one returns the stored pointer while the other one also removes it from the stack. Since it is only possible to work with pointers, it is very convenient to get an element, modify it and then not having to push it back on top of the stack.

Since the actual stack is defined to store and interact with void pointers it can be used generically to store any type of elements, even heterogeneous collections. However, such scenarios hide the possibility for many errors and bugs; this is why it was seen as best practice to create a set of wrapper functions to take care of interacting with the stack itself, accepting structures of specified type.

E.1 An example: ScopeStack

The most important use of the stack in the project is to keep a set of elements that define and characterize scopes encountered while parsing. There is one type of structure used as element, a scope, that holds three fields:

- **braces**: used to count opening and closing braces. At an initial value of 0, it is incremented for every opening brace and decremented for every closing. A scope cannot be closed is this value is not zero. Also a negative value means the respective positive number of excess closing braces.

- **state**: this keeps a reference to the scanner’s state before this scope was opened, to be restored at closing.

- **xclusive**: a flag identifying scopes which should not allow a lookup to iterate up to their parent scope. An example would be the declaration of a struct.

- **id**: this is the numerical identifier of the scope.
F

WCC: The wrapper script

As mentioned from the start of this report, an important part of this project is a simple python script, called WCC, which manages automatically the whole process of compiling WOOL C to machine code, producing a single binary executable. The discrete steps for accomplishing this are as follows:

1. Prepend each source file with a directive including the static WOOL header file.
2. Call GNU CPP to preprocess each source file.
3. Call WPP to transform all source files to GNU C, simultaneously outputting their collective analysis results.
4. The aforementioned information is used to produce a single temporary dynamic WOOL header file.
5. Prepend each source file with a directive including the dynamic WOOL header file.
6. Call GNU CC to compile all source files and link them with the pre-compiled WOOL object file.

The static WOOL header file is the only one including any external dependencies and is designed to be independent of the actual project being processed. So, it can be merged before any other processing step, eventually allowing the preprocessor to have the complete list of external dependencies at its disposal simultaneously. This way any issue with including the same header files multiple times is negated.

On the contrary the dynamic WOOL header file requires for the complete source code to be scanned and analysed before it can be generated. However, the GNU CC will preprocess each source file again, handling this way all the macros and other directives which will exist.

Although the list above describes WCC as using the GNU family of compilation tools, it is only the default behaviour. A configuration file is provided that allows using any other collection of tools. It is very important to keep in mind that WPP supports only GNU C, meaning ISO C and a collection of GNU extensions. These extensions are common between other compilers, though it is always possible that they might not follow the exact syntactic and semantic implications of their GNU counterparts.

Finally, this script accepts plenty of arguments that configure its behaviour or that of its corresponding components. Except the mandatory -f argument which defines the source file list, the rest can be divided into three categories:

```
Usage:  wcc -f <file1>[,<file2>[,...]] [-d:tsar Hv] [-- <GCC arguments>]
Options:
  -f, --files  Comma delimited list of files
  -d, --directory  Make <directory> the output path for all generated files
  -t, --debug  Enable debug verbosity mode.
  -s, --parser  Debug the parser.
  -a, --ast  Debug the AST.
  -r, --printer  Debug the printer.
  -H, --help  Print this message.
  -v, --verbose  Increase verbosity.
```

Figure F.1: WCC usage information.
groups. The first sets parameters of WCC itself. The second group enables verbose output and debugging features of WPP. Finally, the third group of arguments is blindly passed to GCC and has to be prefixed with a double dash \(--\). This last group is very useful for a variety of situations, like linking with an external library; Math (\textit{-lm}) is the most common shiny example, with the default pthreads following right after. Actually the latter is a WOOL's dependency, so WCC is configured to add it anyway. Figure F.1 provides the complete listing of these arguments.
Bibliography


