Department of Mathematics
and Computer Science

University of Southern Denmark

Bachelor Thesis in Computer Science

A Compiler for the Juhl language

Author
William Juhl
wijuh20@student.sdu.dk

Supervisor
Kim Skak Larsen
Professor

June 1, 2023

Abstract

Danish Denne rapport går igennem alle trinnene til at udvikle en compiler der er nem at vedligeholde. Dette opnås ved at opdele compileren i modulære faser der nemt kan udskiftes eller forlænges med andre komponenter. Rapporten gennemgår udviklingen af værktøjer som Flex og Bison og hvordan disse sættes sammen til at danne frontend delen af compileren. Ved brug af visitor pattern samles data omkring programmet ind, som bruges til at konvertere kilde koden til assembly. Til sidst gennemgår rapporten en implementation af en Garbage Collector, som følger med sproget.

English This report covers all the steps involved in developing a maintainable compiler. This is achieved by dividing the compiler into modular phases that can be easily replaced or extended with other components. The report discusses the development of tools such as Flex and Bison and how they are combined to form the frontend part of the compiler. By utilizing the visitor pattern the compiler gathers data from the source program and converts the source code into assembly. Finally, the report presents an implementation of a Garbage Collector that accompanies the language.

Contents

Chapter 1

Introduction

The goal of this project is to develop a compiler for a simple language without the use of external libraries or tools. The compiler emits X86-64 assembly with the AT&T syntax. This means that to run the compiled assembly program the use of an assembler and linker is needed. The language is accompanied with a garbage collector which is written in *C*.

One might wonder why the use of third party tools such as Bison or Flex were not used, and the answer to that is simple: *Where would the fun be in that?* Thus, for this project we will develop our own tools.

Let us first go through the different phases of the compiler pipeline and discover what needs to be developed: A compiler is divided up into two main phases. The front end, and the back end of the compiler. The front end defines the syntax of our programming language. Things such as reserved keywords, what are legal variable names and in what order keywords and symbols should appear.

- The first step of the frontend is to divide the source code into tokens This step is called tokenization. Here we define which keywords are reserved for the language and which variable names are legal and which are not. Flex is a tool that solves this goal, so we will try to make one similar to that.
- The second step is where we will need a parser. Here we take the list of tokens created from the privious step, and combine them into a parsetree. The output of this step is an abstract syntax tree. A tool for generating such a parser is called Bison.

The backend of the compiler defines the semantics of the language. What

happens when the user makes a function call, should it spawn a new thread and make that thread execute the function while the main thread continues execution, or should something happen anytime the value 42 is computed. The backend allows us to define what should happen at any time during runtime:

- In the first step of the backend we take the raw abstract syntax tree and remove any unnessecary artifacts from the parser. This results in a clean AST which is easier to work with.
- The compiler then traverses the AST to collect different information about the program it is about to compile.
- With the collected information it constructs intermediary code.
- Lastly the intermediary code is converted into assembly code.

Language Semantics and Syntax

For this simple language there exists a few keywords, which we will dive into now.

- **let**: This keyword declares that a variable is about to be declared.
- **type**: Indicate that a new type is about to be declared. In C this is the same as a struct.
- **if**: If indicates that we are about to check if some expression is true and if it is execute the following block.
- **else**: If the if statement fails execute the following block.
- **return**: Return from a function call with a given value.
- **int**: A type representing whole numbers using two's complement.
- **print**: Prints a string the std out.
- ***: Indicates that the field or variable should be allocated on the heap.

```
1  let fib: (a: int) -> int;
2  fib = (a: int) -> int {
3      if ( a == 1 ) {
4          return 1;
5      } else if ( a == 2 ) {
6          return 1;
7      }
8      let fib1: int;
9      let fib2: int;
10     fib1 = fib(a-1);
11     fib2 = fib(a-2);
12     return fib1 + fib2;
13 };
14 let n: int;
15 n = 7;
16 print( "The %th fib number is: %\\n", n, fib(n) );
```

Fibonacci Example

In the above example we see an implementation for computing the nth fibonacci number. The first line is a declaration. Here we declare that there exists a function called fib which takes a single integer argument and returns an integer. The second line assigns the function body to the function declaration. Lines 3 through 7 contains the base case. Line 10 through 12 computes the two previous fibonacci numbers in the sequence and adds them together resulting in the nth fibonacci number.

Chapter 2

Tokenization

The goal of the first step of our compiler, the tokenizer, is to convert the source code into a list of tonkens. Ideally we want it to be simple to add new keywords to our language to ease maintenance later down the road. For each recognized string we want to construct a token which stores some metadata about where in the file the token was found and what kind of token it is. The metadata will be used for notifying the user of any potential syntax errors.

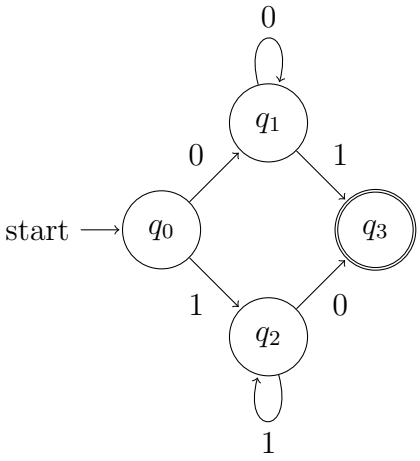
One way to solve this problem is to split the input after each space and then having a bunch of checks that determine what kind of token we are dealing with. This, however, can be expensive computational wise: for a single token we will in the worst case have to check each reserved keyword, and if none of these match, determine if it is a string, number or a variable name. With this approach some maintenance also follows. We can do better.

Flex uses regex to describe which strings match selected tokens, and returns a list of tokens based on what it matches. Let us create a tool similar to this.

2.1 DFA

We first need to know some theory of deterministic finite automata. A DFA is a collection of states, that for each state has arcs that links it to other states. The machine takes a symbol form the input one by one and finds the corresponding arc to get to the next state. The deterministic part of the name, comes from the fact that for each state it has as many arcs as there are symbols in the alfabet resulting in the machine always knowing which arc it should traverse. The alfabet is a collection of characters or symbols that the state machine knows about. To make the machine finite we need to

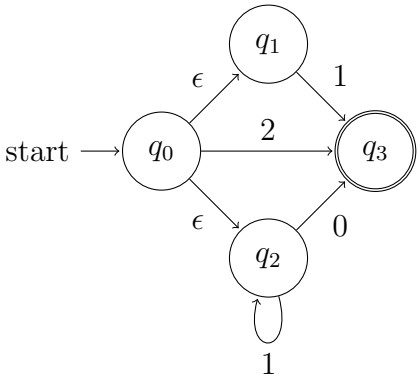
mark some of the states to be accepting, meaning that when it reaches such a state it is allowed to stop or continue if more symbols follow. An example of a DFA is shown below.



The above DFA accepts any input where it either starts with any number og 0's (atleast one) and ends with a 1, or starts with any number of 1's (atleast one) and ends with a 0. The alfabet for the above DFA is $\{0, 1\}$.

2.2 NFA

The difference between a non deterministic finite automata (NFA) and a DFA is that NFAs allow the use of ϵ arcs. ϵ arcs allows the machine to be in multiple states at the same time. If the machine comes across a state containing ϵ arcs it may choose to cross that arc. Furthermore a state can have zero outgoing arcs or even multiple of the same symbol. Hence the non deterministic part of the name. NFAs allows us to both be lazy while diagramming a machine and easier build much more expressive machines. An example can be seen below:



Notice the state q_1. It doesn't have any outgoing arcs with a 0. And the start state q_0 contains two epsilon arcs going to q_1 and q_2. This means that from state q_0 the machine can choose to progress through any of its epsilon reachable states or the state itself. The above machine accepts the following inputs:

$$\{0, 1, 2, 10, 110, 1110, \dots\}$$

2.3 NFA to DFA

In this section I will show that any NFA can be converted to an equivalent DFA through a series of steps.

It is important to remember that NFAs can be in multiple states at any given point. For this reason we will collaps some states into a single state simulating the NFA being in multiple states at once.

Given a set of states we can compute the epsilon closure for that set by:

1. For each state in the set: traverse each epsilon arc and add the reached state to the set.
2. If the newly added state also contains epsilon edges go to (1)

Initially we start with the epsilon closure of the start state. Add this state to a queue and do the following until the queue is empty.

Extract a state from the queue and check each of the outgoing arcs. This gives us the following two cases

1. Each arc has a unique symbol
2. Some arcs has the same symbol

The first case is the simplest one. From the current state add an arc to the epsilon closure of the reachable state.

For the second case we collaps the states reachable with the same symbol and add an arc with that symbol to the epsilon closure of the collapsed states.

Add the computed state to a queue to be processed and repeat.

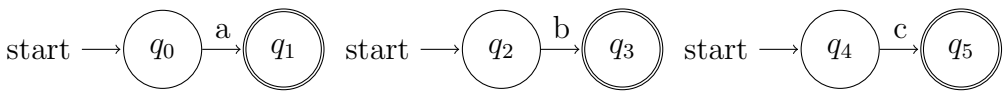
2.4 Regex to NFA

Regex is a language that allows us to make patterns that match quite complex inputs. And luckily for us, DFAs are able to express regular languages. For our purpose we need the following features from regex: concatenation, union and the star operator. We will now show that DFAs are closed under these three operators.

2.4.1 Concatenation

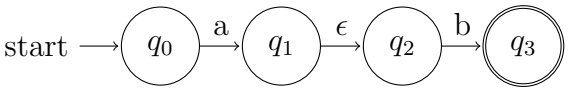
Concatenation is a way to specify that a symbol must be followed by another symbol. Concatenation is usually denoted by the symbol $\circ$ in between two other symbols. Let us imagine the following regex pattern: "a$\circ$b$\circ$c". This pattern will only match if we first see an a followed by a b and ending with a c. The symbol $\circ$ is usually omitted because it is easier for humans to read the following "abc" and recognizing that the pattern only matches "abc".

To show that NFAs are closed under concatenation let us imagine the above scenario of a machine matching the language "abc". We can split the input into three different NFAs each containing a start and a end state. From the two states an arc is connecting them with either "a", "b" or "c". Machine A can be concatenated to machine B by for each finish state in A add an epsilon arc to the start state of machine B. And lastly remove each accepting state in machine A. This will result in a machine AB. After this we can apply the same logic to machine AB and C, which results in machine ABC.



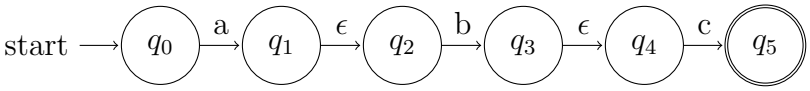
(a) Machine A (b) Machine B (c) Machine C

Initial setup for the three machines



(a) Machine AB

Construction of machine AB

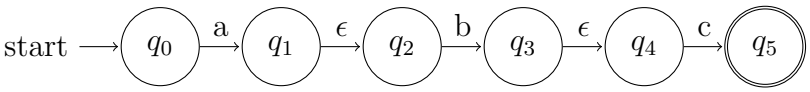


(a) Machine ABC

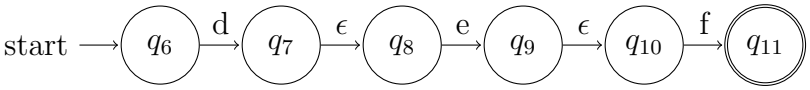
Construction of machine ABC

2.4.2 Union

The union operator allows us to match either the first pattern or the second pattern. The operator is usually defined with a "|" symbol. In many programming languages this symbol is used as an *or* operator. Let us expand on the above example, let us imagine that we would like to match on patterns that either contain "abc" or "def". This would be denoted by the following regex string: "(abc) | (def)". The two machines, *left* and *right*, can be combined by adding a new start state and adding epsilon arcs to the start states of *left* and *right*. To make it simpler to understand let us also add a new accepting state. For each accepting state in *left* and *right* add an epsilon arc to the new accepting state, and remove any accept states in *left* and *right*.



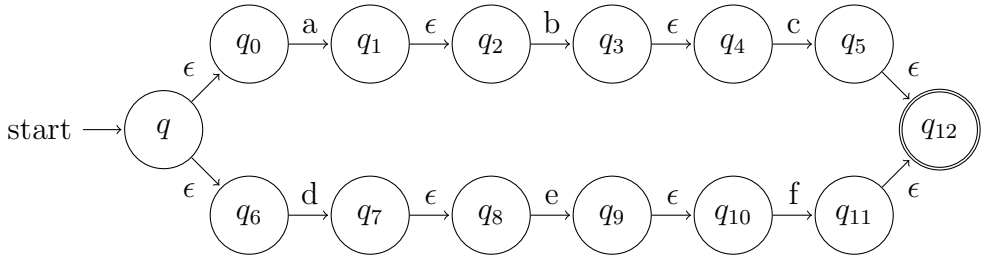
Machine left



Machine right

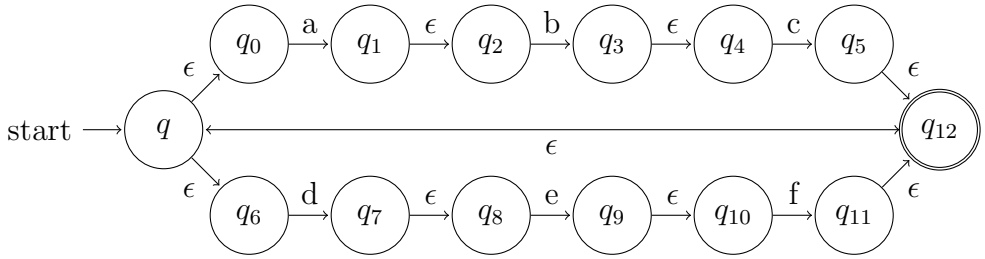
2.4.3 Star

The star operator, usually denoted with this symbol "$*$", lets us match on a pattern recurring zero or multiple times. This can be accomplished by adding an epsilon edge (an edge goes both ways) from the start state to each accepting state in the machine. This allows the machine to accept on the



Union of Machine left and Machine right

empty input and also if the pattern repeats itself multiple times. If we take the above example the regex pattern would look like this "((abc) | (def))∗"



Star operator on the previous Machine

Beyond being able to convert regex to NFAs, we also need to be able to parse the regex language. This can be done with a recursive descent method, however, in the next section we will be building a parser generator, that can do this for us. So let us postpone this for later.

Chapter 3

Parsing

The next major step in the compilation process is processing the tokens received from the previous step and combine them into an abstract syntax tree (AST). This AST contains the semantics of the program to be compiled, and by traversing the tree we will be able to emit the target language.

The two main ways of writing a parser is by either utilizing a parser generator or by writing the parser by hand. This can be done by using a recursive decent technique. Depending on the situation the latter technique can have its merrits. It will provide a much finer grained control over what is possible to parse. The error messages can also be super precise. Essentially, writing a parser by hand the compiler author is able to parse any imaginable language. However, maintaining such a beast can quickly become a nightmare as the language grows and new syntax needs to be added.

Which brings us to parser generators. Tools such as *yacc* or *Bison* generate parsers capable of parsing LR (1) and LALR (1) languages by taking as input a Context Free Grammar (CFG). By utilizing a parser generator we will gain greater flexibility maintaining the compiler and further expanding the acceptable language by modifying the CFG, which is the main advantage of going with this approach. However, the parse table generated for my language takes an enormous 51 megabytes.

There is a third way, which involves using both a parser generator and extending it by hand. This can provide much finergrained error messages to the user, as the compiler author is able to analyze the current state of the compiler at the point of failure, and figure out what went wrong by backtracking the call stack or by looking at the upcomming nodes.

For this project I will be using a parser generator. However, I will not be using an already available parser generator. I will be making my own, which I have called *Parsley*. Parsley is capable of generating LR (1) parsers which fits the purpose of this project. The parser generator will solve two problems for us. Firstly, it will be used to generate a parser capable of accepting regular languages for the tokenizer. Secondly, it will generate a parser for the parsing step, which will, as stated earlier, combine the tokens from the tokenizer into an AST.

In order to create *Parsley*, we first need to understand some of the underlying theory *Parsley* is built upon.

3.1 Context Free Grammar

Cotext Free Grammars are a formal way of describing a language. CFG consists of *terminals* and *non terminals*. Every grammar contains a start symbol, which is usually denoted *S*. The start symbol is a non terminal, which means that it must contain at least one reduction rule. A reduction is what happens when a list of symbols match the given rule, and is able to be reduced into a non terminal. The symbols in the reducion rule can be non terminals or terminals. The notation for such a rule can be seen below.

$$S \rightarrow aBBa$$
$$B \rightarrow b$$

In the above example there are two reduction rules. One of which reduces into S and one into B. The arrow indicates that the left hand side of the arrow can be described as what lies on the right hand side of the arrow. Traditionally small letters indicate terminals and capital letters indicate a non terminal. From the above CFG, we see that the non terminal B can only be expanded into a b, or if we look at it in reverse, the terminal b can be reduced into the non terminal B. Thus the CFG can accept the language $\{abba\}$.

If we take a step back and look at CFGs with the context of compilers, the terminals are the tokens provided from the tokenizer, and the non terminals are the building blocks that combine the terminals into a meaningful tree representation. A rule for accepting an assignment statement could look like the following:

$$\text{n\_ASSIGNMENT} \rightarrow \text{t\_id t\_eq n\_EXPRESSION}$$

Here we can see that in order to make a node symbolizing an assignment we need three symbols. Two of the symbols are terminals, indicated by the t, and a non terminal, indicated by n, called *EXPRESSION*. This non terminal can be anything that the language designer sees fit as an expression. Let us say that in this context free language an expression is a literal number. We would then have the following language:

$$\begin{aligned} S &\rightarrow \text{n\_ASSIGNMENT} \\ \text{n\_ASSIGNMENT} &\rightarrow \text{t\_id t\_eq n\_EXPRESSION} \\ \text{n\_EXPRESSION} &\rightarrow \text{t\_literal} \end{aligned}$$

Then imagine the parser receiving the following stream of tokens:

$$[\text{t\_id}, \text{t\_eq}, \text{t\_literal}]$$

The parser would scan the tokens from left to right one by one, and check against any potential matching rules. It is first when we reach the literal that the parser finds a match from literal to expression. The parser will then replace the terminal with the corresponding non terminal. We will then have the following:

$$[\text{t\_id}, \text{t\_eq}, \text{n\_EXPRESSION}]$$

Now the the second rule matches the input, and we can reduce the input to the following:

$$[\text{n\_ASSIGNMENT}] \rightarrow [S]$$

We now have an idea of what we want to build. We want to build a machine that takes some CFG and converts this into another machine that accepts the language of the given CFG. This machine is the actual parser that will be used for the parsing step of the compiler.

We will now dive into the theory necessary constructing such a machine. To build the parsetable we need to understand three concepts: NULL, FIRST and FOLLOW.

3.2 NULL Set

The name NULL comes from the fact that some non terminals are nullable. Nullable symbols are non terminals that have some rules where the entire right side of the production rule can be replaced with nothing, which indicated with an ϵ. Imagine the following rules:

$$
\begin{aligned}
&S \rightarrow ABC \\
&A \rightarrow AC \mid B \\
&B \rightarrow b \mid \epsilon \\
&C \rightarrow c
\end{aligned}
$$

Looking at the third reduction rule we recognize that the non terminal B can either be achieved by reading a b from the input or nothing at all. This means that the non terminal B is nullable, because it can be replaced with nothing. By transitivity this means that A is also nullable, because A has a reduction rule consisting of entirely nullable symbols. Thus for the above CFG we have the nullable set:

$$\{A, B\}$$

3.3 FIRST Set

The first set is the collection of terminals that can appear as the first symbol in the reduction rule for each non terminal. Thus, the first set for the above CFG will have four collections of terminals, one for each non terminal: S, A, B and C. However, we must remember that some of the non terminals are nullable. This means that FIRST(A) contains both b and c. This is due to the fact that A itself is nullable, and thus the first reduction rule for A can be rewritten as follows:

$$enouA \rightarrow C$$

From what we know now, we can compute the first set for each non terminal:

- FIRST(S) = $\{FIRST(A)\} = \{b, c\}$
- FIRST(A) = $\{FIRST(B), FIRST(C)\} = \{b, c\}$

- FIRST(B) = $\{b\}$
- FIRST(C) = $\{c\}$

3.4 FOLLOW Set

The follow set is the collection of symbols that may appear after a non terminal. Just like the first set it contains four collection of terminals. From the above example we would get:

- FOLLOW(S) = $\{\}$
- FOLLOW(A) = $\{FIRST(C), FIRST(B)\} = \{b, c\}$
- FOLLOW(B) = $\{FIRST(C), FIRST(B)\} = \{b, c\}$
- FOLLOW(C) = $\{FIRST(B), FIRST(C)\} = \{b, c\}$

In the above $FIRST(C) \in FOLLOW(A)$ because the second production rule contains a C just after A and because in the first production rule A is followed by BC where B is nullable. $FIRST(B) \in FOLLOW(A)$ because the in the first production rule A is immediatly followed by B.

3.5 Constructing a Parse Table

Using the above three sets, we can construct a parse table which will be used to parse some input given a grammar. On success this will result in an AST. The parse table is conceptually a state machine, where each arc is a symbol as defined in the grammar. However, it is not a DFA because this state machine only contains arcs that comply with the language as defined by the grammar. The parse table is constructed by running through two phases until no new states can be added. The first phase computes the closure of the given state. The second phase branches from a given state to the next state with a symbol.

The start state of this machine starts with the start symbol and it's production rule of the grammar. Using the following grammar:

$$
\begin{aligned}
S &\to E\$ \\
E &\to T + E \\
E &\to T \\
T &\to x
\end{aligned}
$$

Figure 3.1: CFG

We will get the following:

$$S \to [\bullet E\$, \{?\}]$$

The dot indicates which symbol the parser expects to see next, if it were to reduce using that production rule. The dollar sign is an End Of Parse (EOP) symbol indicating that the parser has successfully read the entire input. The questionmark is the lookahead symbol which is the symbol the parser expects to see if this given production rule's reduction was chosen. Since nothing is supposed to come after the the start production, it is marked with a questionmark indicating that nothing is expected after.

Let us now dive into the first phase. The goal of this phase is to include all the rules from the grammar that this state can produce. For each production in the state take the symbol just right of the dot and if it is a nonterminal add the production rules of that nonterminal. This will give us the following initial state:

$$
\begin{aligned}
S &\to [\bullet E\$, \{?\}] \\
E &\to [\bullet T + E, \{\$\}] \\
E &\to [\bullet T, \{\$\}] \\
T &\to [\bullet x, \{+, \$\}]
\end{aligned}
$$

State 1

The lookahead symbol has for the added entries changed to a dollar symbol for the second and third entry and for the last entry it contains both the plus symbol and the dollar symbol. The algorithm assigns the lookahead for each new entry by computing the FOLLOW set from the entry from which it was added. However, from the parsers perspective everything to

the left of the dot in the entry has already been parsed. Therefore, from the parser's perspective, everything up until the dot is NULLABLE. The last entry contains two lookahead symbols. This is due to the fact that the T symbol has been added from entries two and three. The lookahead from entry 2 is $\{+\}$ and the lookahead from entry 3 is $\{\$\}$.

Now that the closure phase has finished the second phase begins. For each entry in the state make an arc, traversable using the symbol just right of the dot, to a new state. From the old state copy each entry that contains that symbol to the right of the dot to the new state. From here advance each entry's dot by one to the right. The last step in the second phase is to goto the first phase with the new state. This will give us the following states:

$$S \to [E \bullet \$, \{?\}]$$
(a) State 2

$$E \to [T \bullet +E, \{\$\}]$$
$$E \to [T\bullet, \{\$\}]$$
(b) State 3

$$T \to [x\bullet, \{+, \$\}]$$
(c) State 4

The above grammar produces a total of six states, *all are not shown*. From these six states we can produce the parse table. For each state there is an action that can take place depending on the current symbol from the input.

The parser uses a stack to keep track of symbols it has already checked.

1. **Accept**: When reaching this the parser is in an accepting state which terminates the parser and returns the AST.
2. **Shift(i)**: This action eats a symbol from the input and moves it to the stack. Then it does a Goto(i).
3. **Goto(i)**: This action changes the parser's state to state i.
4. **Reduce(k)**: This action tells the parser that it should reduce using rule k. By doing this it pops n states from the stack and push the left hand side of the rule to the stack, where n is the number of symbols on the right hand side of the rule.
5. **Syntax Error**: The parser stumbled across an unexpected symbol. It expected one of the symbols with any of the above four actions in the current state.

Now that we know what each action does, we can go through the rules for deciding which action to populate the parse table with. For each state look

through each entry and decide which one of the four possible conditions are met:

1. If the symbol to the right of the dot is a dollar, put an accept action.
2. If the symbol to the right of the dot is a terminal, put a shift action and let i be the id of the state reachable by traversing using that symbol.
3. If the symbol to the right of the dot is a non terminal, put a goto action and let i be the id of the state reachable by traversing using that symbol.
4. If no symbols are to the right of the dot, add a reduce action with the k'th rule for each symbol in the lookahead.

By following the above directions, we can produce a parse table for the above given grammar:

State/Symbol	x	+	$	E	T
1	s4			g2	g3
2			a		
3		s5	r2		
4	s4			g6	g3
5		r3	r3		
6			r1		

Parse Table constructed from CFG 3.1

If at any point the parser sees a symbol that has an empty cell, it will produce a syntax error.

3.6 Parsley

Parsley is the finished parser generator capable of producing an LR(1) parser given any CFG. Using Parsley we can construct a parser for the tokenizer capable of parsing Regex. By providing the Regex parser the syntax of the language we want, it will return a DFA capable of tokenizing the source code. With this the first step of the frontend is finished. The second step of the frontend is the parser for the actual languange syntax. Just like with constructing a parser capable of recognizing Regex, we provide Parsley with the CFG for the language and it returns a parser for the compiler. These

steps of the compiler chain are only present while developing the compiler. The diagram below shows how the frontend fits together

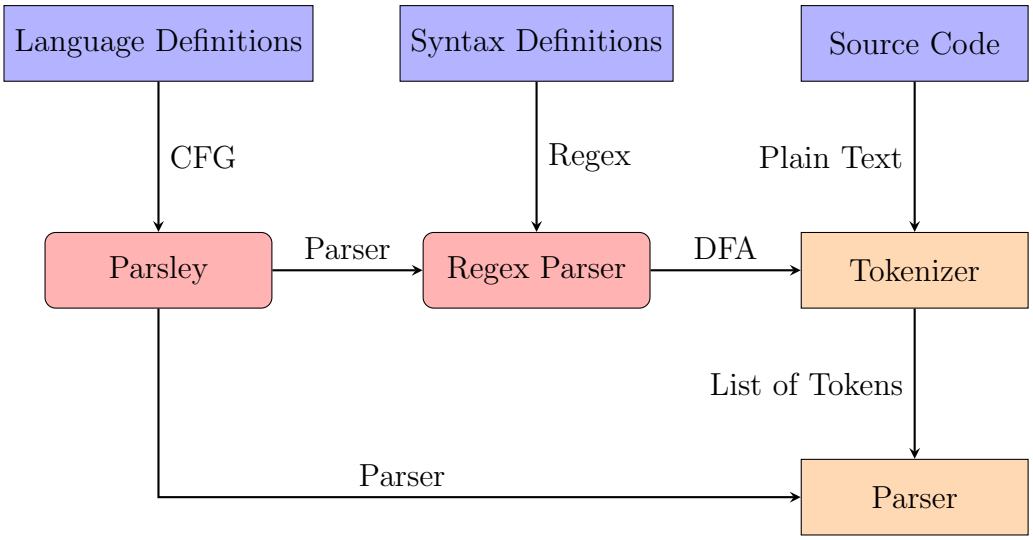


Figure 3.2: Illustration of the Frontend for the Compiler

In the above figure the first two blue boxes are modifiable by the compiler author. Language Definitions define the language's syntax. Syntax Definitions define the syntax for keywords, legal variable names, etc. The Source Code box is the source code the user wants to compile. The red boxes are intermediary programs that are used for constructing the compiler. The orange boxes are part of the compiler and are used in production.

Chapter 4

Compilation

Let us dive into the backend of the compiler. Here we define the semantics of our language. At this stage we have the AST from the parser which we will traverse multiple times in order to collect various components and information about the source code, which determine what should happen at runtime. This stage can be divided into multiple phases just like the frontend. For this compiler the phases are as follows:

1. String Collection
2. Symbol Collection
3. Register The Stack's Layout
4. Register The Structs' Layout
5. Intermediate Code Generation
6. Code Emit

To traverse the AST we will be using the visitor pattern. This allows us to write the logic for traversing the AST once. We can then send different visitors through the AST and execute specific code depending on the currently visited node.

4.1 String Collection

This collector traverses the AST to locate any strings that are used in the source code. A program can quickly use alot of strings, and some of which

are repeated multiple times. For space efficiency reasons we want to only store unique strings in the final output code.

4.2 Symbol Collection

The goal of the symbol collector is to collect the symbols defined in the source code and map them accordingly. Symbols are variables, struct definitions and function names. Variables are usually placed on the stack unless it has been heap allocated. We traverse the AST and collect all declarations one by one.

This phase is also where the compiler checks for scoping rules. I have chosen to implement static scoping, and this means that during an inorder traversal of the AST if at any point a variable is referenced, which have yet to be declared, we have encountered a scoping error. The symbol has been referenced before being declared. Imagine the following scoping example:

```
1 let fun1: ();
2 let var1: int;
3 var1 = 42;
4
5 fun1 = () {
6     let var2: int;
7     var2 = var1 + 4;
8     print("Variable 2 is: %\n", var2);
9 };
```

Static Scoping

In the above example at line 7 there currently are two stack frames. One for the outer scope, and one for the scope of the function *fun1*. In order to read the value of *var1* we first need to go to the outer frame, and then find the correct offset for that specific variable on the stack. For that reason we construct a lookup table to see if we are in the same scope as the referenced symbol. If we are not, we need to locate it. This can be done by following the old basepointer stored on the stack and recursivly check if we are in the correct scope.

4.2.1 Garbage Collector

Arguably a garbage collector is not a part of the compiler, however, the metadata that the garbage collector needs will be collected in the compiler.

For the garbage collector to function we need to register all the pointers located on the stack. Additionally we need to go through all user defined

structs and identify which have pointers and what offset these pointers have from the start of the memory block.

Let us call this for a *Layout*. A layout is a series of bitfields indicating with a one if that specific offset is a pointer, or a zero if it can be ignored. For this we can use 64 bit bitfields, where the first bit indicates if there is another bitfield after this. This is necessary for large structs or stack frames, where there are more than 63 pointers or variables. Imagine the following struct definition:

```
type Thing: {
    let a: *int;
    let b: int;
    let c: int;
    let d: *int;
};
```

Type Definition

In the above example the bitfield for the struct Thing will have the first bit set to zero, because there are less than 64 fields the struct. The second and the fifth bits are set to one and the rest zero.

4.2.2 Stack Layout

We will now discuss how to collect the pointers located on the stack. For each possible stack frame we have to generate a layout which keeps track of where a pointer is located in respect to the currently active stack frame. Stack frames occur in functions and in the main scope. This means that we traverse the main scope to identify its layout, and for each function do the same. For each traversal of the AST we locate any pointer declarations to the heap. Since the symbol collector already is mapping offsets for each local variable, we can use this offset counter to mark pointers on the stack.

4.2.3 Struct Layout

Just like generating layouts for each unique stack frame we generate layouts for each unique struct. Whenever we find a struct declaration we register any pointers it may contain, and depending on whether this specific struct has been allocated on the stack or on the heap, the pointer offsets will be negative or positive respectivly.

4.3 Intermediate Representation

Now that the components needed to construct the emitted code have been collected, we can start to define the semantics of the language. At this point we will convert the AST to IR which is an internal representation of the output code. For this compiler the IR is very close to X86–64 assembly. The IR is divided into an opcode and a list of operands depending on the opcode. Additionally it can contain comments to make it easier to understand what the compiler has emitted. The advantage of going through this step is that later on different optimization strategies can be applied such as peephole optimization, which identifies patterns of opcodes and replaces them with more efficient opcodes. An example of this could be a sequence of a push followed by a pop. This sequence can be replaces by a simple move instruction instead.

4.4 Constructing the Language Semantics

At this point we have some options. We can go through the AST and start by constructing the IR for each function and then assemble the code located in the main scope. This will result in multiple lists of IR code. By choosing this path it will be easier in the code emit phase. Another option is to go through the AST line by line and construct everything inorder. This brings some disadvantages. We have no seperation of the main code and functions. Another huge consideration is parallalization of the IR code construction. With the latter choice it would be very difficult to find different compilation units that can be compiled seperatly. Using the former technique we can divide each function assembly as its own compilation unit and harness the power of the computer's cores. Therefore let us choose that option. Because each function definition is a subtree it is easy to isolate them.

Let us now go through how we can construct different high levels emantics.

- **Variables**: Variables have two actions that can be performed with them. First it is to be able to store some data. Second we can read the value it stores. The symbol collector has already chosen a spot on the stack for each variable, so to read and store values the compiler can refere to that location. If the variable referenced is outside the current scope, the runtime needs to retrieve the old base pointer located on the stack until it has the correct pointer to where the variable is stored. This, however, makes accessing out of scope variables very expensive. For each level of nesting it requires an additional move instruction

whereas before it only required a single move instruction for variables located in the current scope.

- **Pointers**: We tell the runtime that we want some memory on the heap capable of storing the desired type of the variable.
- **Structs**: Structs are a collection of variables located right next to each other. For that reason fields in a struct can simply be thought of as variables with a longer name.
- **Functions**: Functions are seperate blocks of code that can be called throughout the execution of the program. Each function has a label that the program can jump to when a call to that function has been made. For the functions footer it needs a return such that the assembler can clean up after itself and return to the calling code.
- **Branching**: Branching is a sequence of conditions, that when met, executes a block of code. This language supports if statements, if else statements, and if else if statements. This is implemented by placing labels that mark each conditional check and another marker at the end of the if statements. If the first condition fails, the runtime checks the next condition until it has exhausted the list of if statements or if one of the conditions succeed. If the list contains an else statement that block of code will be executed if none of the prior conditions were met. After a successful condition and the execution of the block has finished, it jump to the end of the list of if statements. This ensures that only a single block of code will be executed.
- **Printing**: This language supports string interpolation in print statements. This works by printing the string in slices and inserting the specified input in between the slices. The arguments to be interpolated are evaluated from left to right and inserted in that order into the string.
- **Loops**: While this language does not implement a while loop, it will be easy to implement one by using the same logic as brancing does. Instead of jumping to the end of the while loop on a successful execution of the loop body, it jumps to the guard of the loop. This implements a while loop.

4.5 Code Emit

The last stage in the compilation process is the actual code emit. The emitter first inserts some data fields for storing strings that are used in the program, stack layout pointers and struct layout pointers. All three types of fields could easily be filled in during compile time, however, I have chosen to fill the last two types at the start of the programs runtime. This offers no advantage and should have been done at compile time to avoid unnecesary overhead during runtime. After the fields have been filled out the compiler moves on to emit the code for each function body. For each function it puts a label for that function, a header, the body and lastly a footer. The header contains instructions to store the old base pointer and set up a new base pointer. It also registers a new stack frame with the runtime. The footer removes the stack frame from the runtime and restores the old base pointer. The only thing that remains is the main body of the program.

The main program also contains a header and a footer just like functions. Additionally the main program initializes the garbage collector, and registers the stack and struct layouts. After this the compiler iterates through the list of IR code and converts it to assembly. Because the IR is just above assembly converting it is straight forward. The IR code for adding two registers together looks like this:

```
1 ADD [RBX, RCX]
```

Which would become:

```
1 addq %rbx, %rcx
```

Chapter 5

Runtime Machine

Garbage Collector

The goal of the garbage collector is to reclaim any unused heap allocations. There are multiple ways of doing this such as reference counting which is used in languages like C++ and Rust by using socalled Smart Pointers. However, reference counting has some limitations. Reference counting works by having the runtime count how many variables holds a reference to a piece of memory, and when this number reaches zero, the runtime will collect the unused memory and mark it as usable for reconsumption. Imagine one developing a Tree Datastructure where each node has references to their children and their parents. In this scheme the datastructure would never be reclaimed because the reference count for any of the nodes in the tree never reaches zero.

For this compiler I have chosen to implement a Mark and Compact Garbage Collector. This algorithm works by dividing the heap memory into two blocks. One of which is used for allocating into, and the other is used to compact the marked memory blocks into. This algorithm does not find the unused memory blocks, rather, it finds the used memory blocks and compacts them. This accomplished two things:

1. Cache locality. The used memory blocks lie contiguous next to each other. This is important for performance and avoiding cache misses.
2. Unused memory is located in a single division of the splitted heap and can be marked as available for allocation.

The algorithm can be divided into two steps. The first step is collecting

all the memory blocks which are still being used. The next step is to move the used memory blocks to the other heap section. To find the memory blocks the algorithm traverses the stack and registers all the pointers. For each pointer on the stack it follows the pointer to the heap and finds all the pointers in that memory block. This continues until all the pointers have been traversed. Each of the visited memory blocks are marked. The next step is to move the marked memory to the other heap section, which can be done in two different approaches, either a DFS or a BFS:

- With a DFS approach, cache locality will be dependent on the order of the pointers in the memory blocks. The pointers placed first in the source code would be moved first.
- With a BFA approach, cache locality for arrays of structs with pointers and other data would be packed closer to each other. However other datastrucures such as trees would be spread further apart.

Therefore I have chosen to move the memory blocks with using a DFS approach.

Stack Frames

For each function declared the compiler would also have to create a stack frame for that function. Each time a function is called the runtime generates a new stack frame and stores it for the garbage collector to access. Whenever a function terminates the runtime has to remove that stack frame from it's own stack of frames to ensure that the garbage collector does not touch the stack above the stack pointer.

Heap Blocks

For each allocation on the heap, there is some metadata. This metadata consists of a Layout pointer and a size field which indicates the size for the memory block. So for the type *Thing* from the above example would look like this on the heap:

```
type: LayoutForThing {
    let bitfield: int;
};

type Thing: {
    let layout: *LayoutForThing;
    let size: int;
```

```
    let a: *int;
    let b:  int;
    let c:  int;
    let d: *int;
};
```

Chapter 6

Concluding Remarks

As indicated in the report some features are missing, such as loops, additional types, and arrays. Loops and arrays will be rather simple to implement. Loops are modified if statements with a jump statement at the end of the body that jumps to the guard. If the guard fails it should jump below the body of the loop. Arrays can be implemented as a struct with n number of elements indicated at compile time. Additional types are a bit more complex. To add types the compiler needs a type checker to ensure that only legal operations are performed on the types. Additionally, the compiler needs information about the physical sizes of the types. Currently the compiler assumes that every type is of size 8 bytes.

The parser requires a lot of space on the heap. Currently the parser takes more than 50 megabytes of space. Constructing the parse table for the language takes about ten seconds on my machine, so I have opted to serialize the parse table into a file and load that file for compilation. However, there is still some overhead in loading the parse table from the file. Adapting Parsley to convert the LR(1) parser into an LALR(1) parser would improve on the space requirements and thereby also decrease the overhead of loading the compiler. Alternatively Parsley could be changed by instead of producing a parse table it would construct high level code that can be modified by the compiler auther afterwards.

Java is incredibly slow for this sort of task. For compilers to be enjoyable for the user, the compiler needs to be fast. Because of the interpreted nature of Java and its garbage collector and the overhead of the parser this simply is not possible. Currently the biggest bottleneck in the compiler is the frontend. It has however been fun.

Bibliography