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

May 16, 2023

Abstract

English your abstract in English

Danish your abstract in Danish

Acknowledgments

Thank you, Mom & Dad!

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 our compiler 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 this phase 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 essentially define what should happen at any time in runtime:

- In the third step we take the raw abstract syntax tree and massages it by removing any unnessecary artifacts from the parser. This results in a clean AST which is easier to work with.
- Lastly we go through our generated tree to produce the assembly language.

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 make 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 user errors.

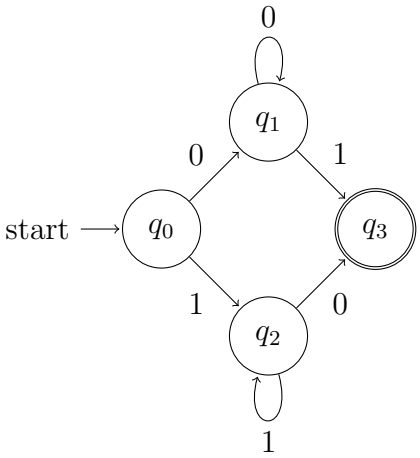
One way to solve this problem is to split the input after each space and then having a bunch of checks that determine what kind of token we are dealing with. This, however, can be expensive computational wise: for a single token we will in the worst case have to check each reserved keyword, and if none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

DFA

We first need to know some theory of deterministic finite automata. As the name implies we are dealing with a machine that is both deterministic and finite, meaning that it always knows what to do, and that it is not infinite. A DFA is a collection of states, that for each state has arcs that links it to other states. The deterministic part of the name, comes from the fact that for each state it has as many arcs as the alfabet. The alfabet is a collection of characters or symbols that the state machine knows about. To make the

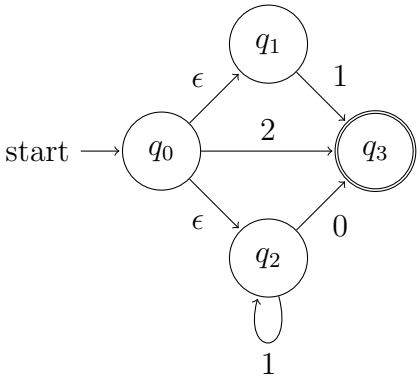
machine finite we need to mark some of the states to be accepting, meaning that when it reaches such a state it is allowed to stop or continue if more symbols follow. An example of a DFA is shown below.



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 $\{0, 1\}$.

NFA

The difference between a non deterministic finite automata (NFA) and a DFA is that NFAs allow the use of ϵ, which indicates that in the current state, we can progress through the arc marked with an ϵ which essentially simulates our machine being in multiple states at any given point in time. Furthermore a state can have zero outgoing arcs or even multiple of the same symbol. Hence the non deterministic part of the name. This allows us to both be lazy while diagramming a machine and 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 essentially 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:

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

NFA to DFA

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

Regex to NFA

Which will be covered in the next section.

Constructing Tokens

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, and by traversing the tree we will be able to emit the target language.

The main two 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 enourmus 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.

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.

NULL

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:

$$\{A, B\}$$

FIRST

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:

$$A \rightarrow C$$

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

FOLLOW

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.

Parsley

Whenever a reduction happens, ie. from the above when a b is reduced into a B we should remove the b from the input and replace it with B. Thus, for each rule that describes the language for the programming language, we need to tell the parser generator what should happen when the specific reduction happens.

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

use to generate IR (intermediate representation). IR is a construct designed to make it easier to manipulate and handle the code during construction. For this project the IR is a set of instructions and

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 functions.

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 in what offset they have from the start of the memory block.

Stack Layout

We will now discuss how to collect the pointers located on the stack. Let us first imagine that no functions has been defined, thus the runtime only has one stack frame. We can then traverse the AST and locate any pointer declarations to the heap and struct declarations that contain pointers 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. 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.

Struct Layout

Intermediate Representation

Code Emit

Chapter 5

Runtime

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 split. 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 split, 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

Chapter 6

Concluding Remarks

Further Improvements and Missing Features

Bibliography

Appendix: Lists of Stuff

This is under development!

These issues are to some extent my personal opinions!

The topics are to a large extent guided by the type of mistakes Danish students would make.

There are of course numerous issues that I do not discuss, some of which are hard for Danes, such a subjunctive forms, rules (other than the obvious) for the use of singular vs. plural forms, etc.

Matters of Style

Miscellaneous

- Id est: use punctuation as ". . . ,␣i.e.,␣". The construction should be followed by more than just one word, usually a whole or partial sentence. The construction should never start a sentence. To be very strict, it should also not start the inside of a parenthesis (use "that is" followed by a comma instead). The same rules apply to "e.g." If you insist on *not* using a comma, make sure to use a backslash to prevent LaTeX from making an end-of-sentence space.
- Do not concatenate words or hyphenate them just because you would in Danish. It is called "type checker", "data structure", "garbage collector", etc.
- Do not start a sentence with "Else"; use "Otherwise".
- Do not start a sentence with "But"; use "However", "On the contrary", or something else appropriate.

- When capitalizing a headline, capitalize both words in a hyphenated construction.
- Capitalize named entities such as "Figure 42".
- Unless one is comparing numbers, small numbers in text should be spelled out.
- There are (almost) always better (more precise) words than big, do, put, and things.
- In formal writing, do not use contractions such as "don't".
- In formal writing, "like", in comparisons, should most often be "such as".
- Do not use "OK".
- "Look at" is informal and should often be replaced by "Consider".
- Use "essentially" very sparingly. Never use it if something is actually exactly as described. Otherwise, it is often possible to explain or hint at the difference with few words instead of using "essentially".
- I see many word order errors such as "the decisions resulted always in more work" instead of "the decisions always resulted in more work".
- "It" cannot be the subject of something such as "it is forbidden to smoke here".
- Do not use "less" for countable items, i.e., "Adam had less apples than Eve" is incorrect and should be "Adam had fewer apples than Eve".

Commas

- Place a comma before "then" or instead of a "then" that has been left out in an "if-then" construction. Also after the first part of "since" constructions and similar (where "then" is of course out of the question).
- Use commas around "however" and similar constructions.
- In lists, use comma after the penultimate item in the list, e.g., before "and". Also use a comma before "etc".
- There are exceptions, but normally there should not be a comma before "that".

- Use a comma after an introductory clause or word, just before a main clause. Main clauses joint by most conjunctions ("and", "but", etc.) should also be separated by commas.

LaTeX

- Use commands for most constructions and variable names other than one letter math variables; see examples in this file.
- Remember backslash after abbreviation periods as "vs.\␣".
- Never typeset names in math mode, i.e., "$left$" is strictly forbidden. Use "left", or just "\textit{left}" if you do not need math mode. This is how ugly is looks if you use math mode: $left$. And this is when it is done correctly: *left*.
- Use ~ (space forbidding line break) between names of numbered entities, e.g., write "Figure~42" to avoid that "42" can appear at the start of a line. (Of course, you would never write "42", but instead refer to a label.)
- Use one hyphen to hyphenate words and two hyphens to create the little line between two numbers of an interval. A dash should be either two hyphens surrounded by space or three hyphens and no space. It is a style issue and a paper should only contain one type.
- Sometimes a style is enforced on you, but if you can decide yourself, I find it much easier to read a text where paragraphs are separated by vertical space than when a new paragraph is just indicated by indentation. It is of course a style matter that should be defined in the preamble.
- Use `\cdots` for missing parts of a sequence when there are no commas; if there are commas, as in $1, 2, \ldots, n$, use `\ldots` to show the continuation.

Writing Advice for TCS

- Think carefully about the naming of objects. When more than one object of the same type is used, it is often a good idea to use letters close to each other in the alphabet. That establishes a sort of typed language that helps the reader. However, skip letters with very estab-

lished meaning, e.g., it is often not good to use f for something that is not a function. Primes can also be used to stay in the same name space. Also using first letters of the names of entities, equivalent Greek letters, etc. can be helpful. And do not give a name to a concept or introduce notation for it unless you need it later.

- Sentences explaining progress in proof can be helpful: Where are we, and what are we doing now?
- Do not say "It is easy to see that ...". If it is true, you can almost always in the same space say "Since this and that, ...".
- Insert data unreduced into formulas and then use arithmetic, so it is easy to see where things come from, i.e., do not combine insertion and reduction steps.
- Be careful with pronouns and be sure that it is clear what they refer to. Very often it is better to write the term "it" is referring to instead of writing "it".
- Things that are the same should be written the same, so that one can more clearly see differences. Technical writing is different from writing literature! Also use a fixed word for any action, i.e., do not use put, place, write, etc. for the action.

Slides for the Presentation

- Use the slides primarily for figures, graphs, and keywords. It is seldom a good idea to include many full sentences (except maybe a theorem statement) and almost never a good idea with full paragraphs of text.
- Think about how you *supplement* your written work, rather than just showing excerpts. For instance, if you show the workings of an algorithm, you can have the algorithm on one side and the result it is working on on the other side, and you can highlight lines of the algorithm as the results change incrementally.