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 13, 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.

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:

- 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.
- 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.

After all of these steps, which are simplified for introductory purposes, we finally have our compiler.

# 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 know 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.

## NFA

## NFA to DFA

## Regex to NFA

Which will be covered in the next section.

# 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. 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.

# Chapter 4

# Compilation

# Chapter 5

# Concluding Remarks

Veni, vidi, vici!

# 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 "$\mathit{left}$", 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.