Physically Based Simulation

Alexander Schlögl

June 1, 2018

Contents

This is **my interpretation** of the lecture slides. I tried to be very verbose and explain everything, all while removing irrelevant parts from the lecture. Using this you should be able to pass the lecture easily. **However, I do not take responsibility for any bad results and will not be blamed from anyone. This was a lot of work and I did it to save others (especially students of following semesters) from having to do this themselves. Use this summary at your own responsibility.** If you have any feedback, feel free to create an issue on the git. I don't promise I will fix anything, but I will try.

1 Introduction to Compilers

A compiler is a program that takes code written in a source language, which is usually a high-level language, and transforms it into a target language, often object code or machine code. In the toolchain that transforms high level code to machine code, there also are other, compile-related programs, which may or may not work together with a compiler:

- **Interpreters & just-in-time compilers** often used for scripting languages (and Java)
- **Assemblers** translate the assembly language into machine code
- **Linkers** combine different object files into executable code
- **Loaders** load shared librabries (relocatable code)
- **Preprocessors** perform macro substitutions
- **Editors** are used to edit the code
- **Debuggers** allow step-by-step execution of the executable
- **Profilers** create memory and runtime profiles of the executable
- **Binary Inspection** allow inspection of the target code in the executable

1.1 Language Description

As a compiler needs to be tailored to the source and target language, describing languages is an essential part of building a compiler. Languages are usually defined at three levels:

- **Lexical level**: The lexical level of a language is defined by a dictionary. The dictionary contains a list of keywords and formats for the different data types, as well as valid variable names, usually defined using regular expressions.
- **Syntactical level**: The syntax of a language is defined by a grammar, describing valid control structures of the language.
- **Semantic level**: This describes the meaning of well-defined sentences in the language, and is often defined (in prose) in the language documentation.

1.2 Phases of a Compiler

A compiler operates in phases, split according to the tasks performed. Common phases of a compiler are shown in Figure 1. While the distinction between the phases is not always clear cut, keeping a degree of modularity is often beneficial.

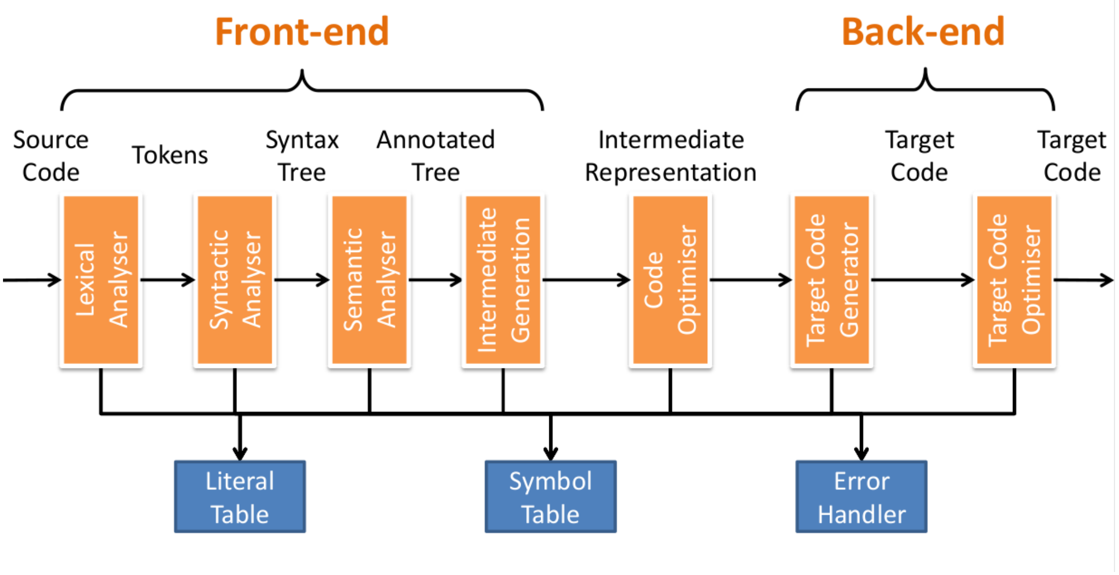

Figure 1: The phases of a compiler

Lexical Analysis (Scanning)

Scanning is the process of taking in a stream of characters and outputting a stream of tokens. This means splitting the source code into variable names, literals, etc. During this phase a compiler can also enter *identifiers* into the *symbol table* and *literals* into the *literal table*.

Syntactic Analysis (Parsing)

During parsing the stream of tokens is used together with a grammar to create a *syntax tree* and report syntax errors to the user.

Semantic Analysis

Semantic analysis checks the meaning of the program, and annotates the syntax tree with *attributes*, e.g. declarations and data types. Some semantics can only be checked while the programming is running (think dynamically allocated arrays), so not all errors are caught here.

Intermediate Code Generation

If multiple source and target languages or platforms are going to be supported it can be very beneficial to generate an intermediate representation that is independent of platform and language. Using an intermediate representation removes the need of creating a compiler for every combination of source and target platform. This reduces the number of parts that need to be written from $m * n$ to $m + n$, where m is the number of source platforms and n is the number of target platforms. The required for adding a new source or target platform also drops from m or n to 1.

Intermediate representations also have the benefit of making optimization through multiple passes easier. A good example of intermediate representations being used is the LLVM compiler.

Code Optimizer

The code optimizer works on the intermediate representation by applying optimizations. An optimization is a transformation that improves performance of the code in one or more metric. Examples are dead code eleminition, consant folding or propagation, etc.

Code Generator

During this phase the actual target code is generated. This can be Assembler, or any other target language. Memory management, register allocation and instruction scheduling are the main challenges here.

Target Code Optimizer

In the last phase optimizations that are target specific are done. This includes replacing instructions with faster ones, data pre-fetching and code parallelization where possible.

1.3 T-Diagrams

A compiler is defined by three languages:

- **Host Language**: This is the language in which the compiler itself runs.
- **Source Language**: This is the language of the input.
- **Target Language**: This is the language the compiler produces.

Any and all of these three languages can be the same. If a compiler produces code in a language that cannot be run on the host machine (the one doing the compilation), it is called a *cross-compiler*.

Compilers are often represented using T-Diagrams, with the letters denoting the different languages. An example is shown in Figure 2.

Bootstrapping

In order to create a compiler for a new language, one can save some work by employing a process called *bootstrapping*. During this process, a compiler for the new language is written in the new language, as it can then make use of the many neat features that were included in the new totally not bloated language that is super awesome and will end all other programming languages (/s). The language creators then write a quick and dirty compiler in a different language. This compiler doesn't have to be powerful, it only needs to be able to compile the "good" compiler. By combining the two we then get a (hopefully) correct, but inefficient compiler. Then we can recompile the "good" compiler with the minimal one to get the final version of the compiler, which can then compile all future versions of itself (until you include new language features). The full workflow is shown in Figure 2

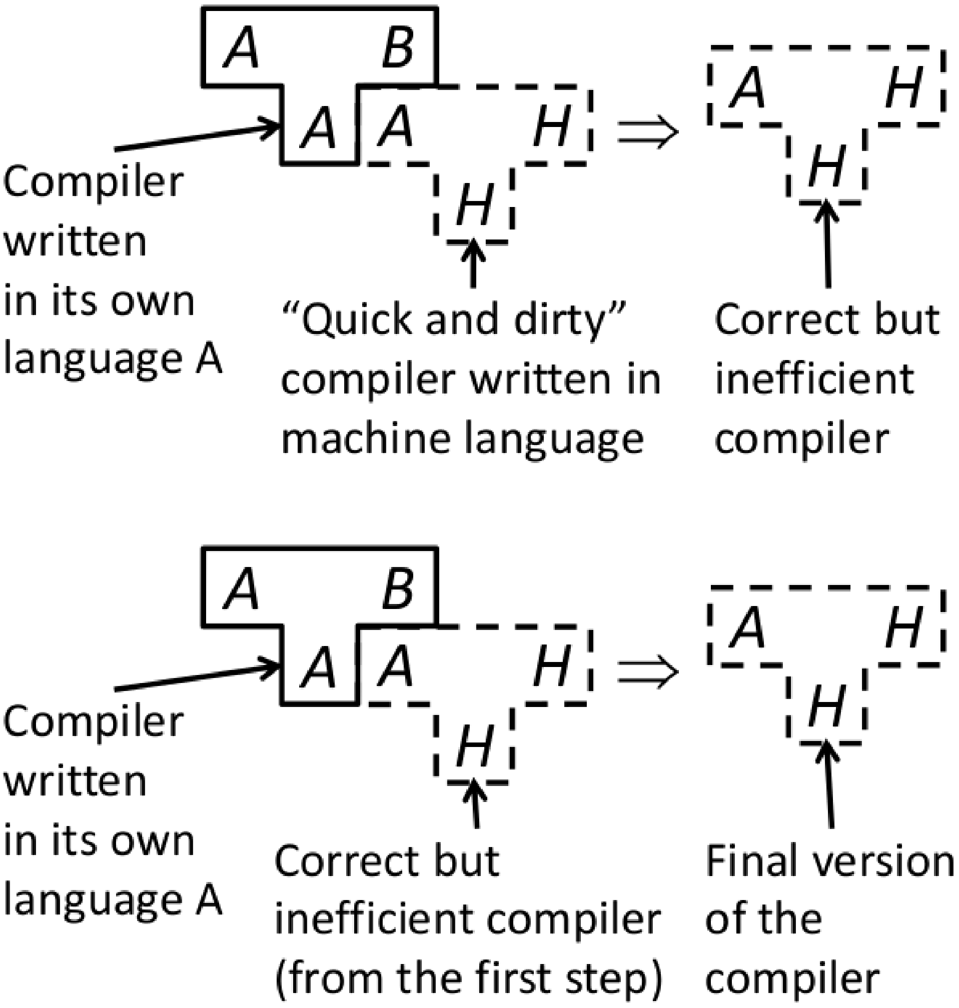
A
B
A
A
H
H
⇒
A
H
H
Compiler written in its own language A
"Quick and dirty" compiler written in machine language
Correct but inefficient compiler
A
B
A
A
H
H
⇒
A
H
H
Compiler written in its own language A
Correct but inefficient compiler (from the first step)
Final version of the compiler

Figure 2: The bootstrapping process in less text and more images

Porting

Porting is the process of moving a compiler written it its own source language A from machine H to machine K. In order to do this, a compiler is written in the source language A with target language K, called a retargeted compiler. This is then compiled with the original compiler and produces a cross-compiler. The cross-compiler runs in language H and produces language K from source language A. The retargeted compiler is then compiled with the cross compiler to create a compiler for language A that runs in language K. The entire workflow is shown in Figure 3

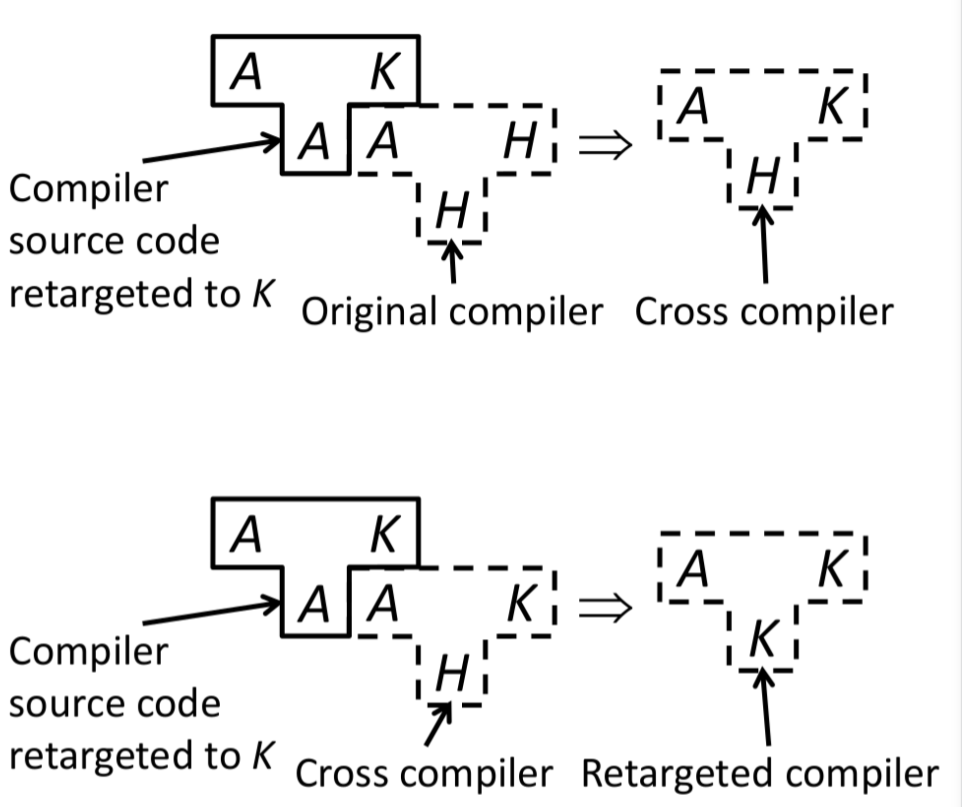

Figure 3: Porting a compiler

Combining T-Diagrams

Combining T-Diagrams is super easy and straight forward. Just replace the language (or letter) to the left of the T-Diagram with the one on the right. A few examples are shown in Figure 4

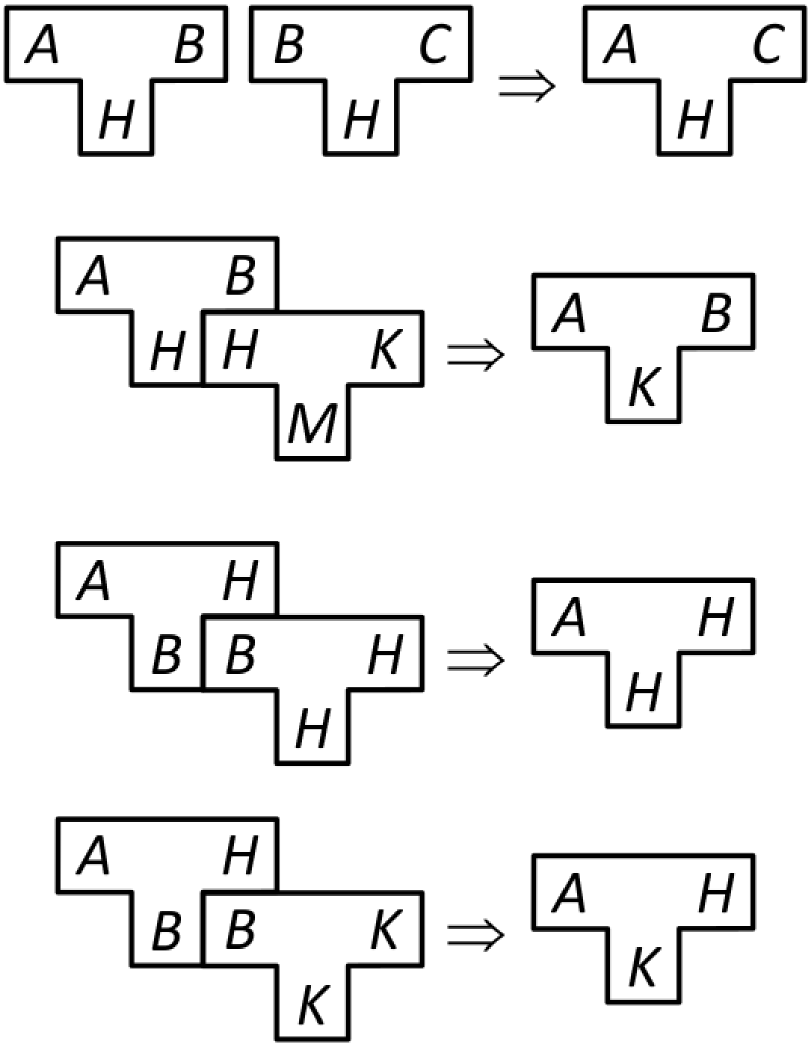

Figure 4: Combining T-Diagrams

2 Lexical Analysis (Scanning)

During scanning we want to split a stream of characters into a stream of tokens. The main goals are separating keywords, variables, constants, operators, etc. We do this by creating regular expressions (regexes) for every type of token we want. These regexes are then converted to finite automata, which we can simulate nicely. We also want this phase of the compiler to be as efficient as possible, in order to save time for the more complex phases.

2.1 Tokens

Tokens in general are defined as an enumerated type, because you almost never define new keywords.

> If you are using a language where you can define keywords dynamically, seek help.

Common token types are keywords (if, else), special symbols (+,-,*), identifiers (for variables) and numbers (int, float, ...). These tokens can have one or more attributes, e.g. the string matched for the token (*lexeme*), the numerical value for numbers, the operator for special symbols, ... These attributes, as well as the type, are stored in a token record, which is just a simple container object or struct or whatever your language supports. The scanner is usually controlled by the parser, which just calls it again and again, each time receiving a single token and processing it (adding it to the token table, matching grammar rules, ...).

As most of us are in computer science because we are both lazy and smart (being less dumb than the average is sufficient), we want to do as little work as possible, and because some people actually program for fun, we don't have to. All we have to do is specify the different program tokens with regexes, and let an automatic tool build the lexical analyzer for us.

2.2 Regular Expressions

I will just skim regexes here because honestly, if you don't know them by now you have been cheating on all your CL exams, so why are you even reading this? All you need to start is an alphabet Σ, and the rest works as defined by this very simple grammar:

$$E = a \in (E \cup \{\epsilon\}) \mid E|E \mid EE \mid E^* \mid (E)$$

So regexes can do:

- letters
- or
- concatenation
- repitition
- subexpressions

as defined in this order. The notation for regexes is quite inconsistent, but this is the bare minimum, as defined in theoretical CS. Most tools that use regexes can do some syntactic sugar, namely $+$ for one or more repititions, . for any character, $[a - z]$ for any character (works with numbers as well), $\sim$ for set negation (ˆ for character classes) and $?$ for optional subexpressions. With this you can create most regexes quite comfortably.

Ambiguity

Some strings may be matched by several regular expressions. Usually the longest match is used (e.g. "forever" is not a keyword, even though it starts with "for") but sometimes even that fails. If there is doubt, most scanners will just use the rule that comes first, but for some more granular decisions token delimiters can be needed (e.g. line end or whitespace). However, the problem with these is that they should not be consumed but instead returned to the input stream. This is called a lookahead character. Usually one lookahead character is enough, but sometimes more is required.

2.3 Deterministic Finite Automata (DFA)

Seriously, you should know this. For an actual implementation, you just use a switch (or a series of ifs if you have to) for the incoming character and then go to the next state accordingly. You can also do the same thing with a transition table, and code less, but transition tables can get very large. Then you have to decide whether you want to implement sparse matrices (or use them if you can) or write a whole bunch of stupid switches.

2.4 Constructing DFAs from Regexes

This works just as it did in FLAT (or FML). You create an NFA from the regexes (usually by combining NFAs for the subexpressions), and then create a DFA from that. There's really not much to it, but if you're still unsure, just look at the FLAT slides, or ask you friends.

3 Top-Down Parsers

> The following section is not part of the exam. I still think it's interesting, and so it's part of this summary. **EDIT:** Well... at least I thought it's interesting, but as soon as you get to LL(1) parsers and their creation everything just goes downhill. So yeah, don't bother reading this chapter, I just included it because deleting it would mean wasted work.

Top-down parsing starts with the first symbol of a token stream and follows the steps of a leftmost derivation of the grammar. These parsers traverse the parse tree of the program in pre-order form from the root to the leaves. As the next choice of derivation can sometimes be ambiguous, multiple methods of disambiguation exist. One is using *lookahead tokens*. By looking ahead in the token stream far enough to find a point where the rules become disambiguous, the parser can always find the correct derivation. This method causes some memory overhead, as the lookahead tokens have to be saved somewhere.

The alternative is using *backtracking*, and just deriving according to any possible rule, and backtracking if the choice was incorrect. This method potentially causes additional computation time due to the backtracking. This computational overhead makes them too slow to be used for practical parsers.

3.1 Recursive Descent Parsers

A recursive descent parser follows a very simple procedure. It takes the next input token, compares it to the expected ones for all possible rules, and recursively traverses down the parse tree for the *one possible* rule. The matching for Nonterminals is done in a specific function for each non-terminal, and the terminals are matched in one large function. Figure 5 shows a grammar and pseudocode for the corresponding recursive descent parser. The procedure (a function without a return value) for matching the *factor* non-terminal is on the left, and the match procedure is on the right.

```
TokenType token ;

procedure factor () ;
begin
   case token of
   ( :      match ( ( ) ;
            exp () ;
            match ( ) ) ;
   number : match (number) ;
   else     error () ;
   end case ;
end factor ;
```

```
exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → ( exp ) | number


procedure match ( expectedToken ) ;
begin
if token = expectedToken then
   getToken () ;
else
   error () ;
end if ;
end match ;
```

Figure 5: Pseudocode for an arithmetic recursive descent parser

Due to the way recursive descent parsers work the grammar has to fulfill certain properties:

- The next possible rule has to be unambiguous. This can be achieved by left factoring the original grammar.
- The grammar may not contain left recursion, as that would result in an infinite loop in the parser. This can be solved by left recursion removal.

Left Recursion Removal

For this we discern between two cases: direct and indirect left recursion. Direct left recursion is when the product of the rule starts with the nonterminal itself ($A \rightarrow A\alpha$). Indirect left recursion is when it is possible to derive a string of nonterminals and terminals that start with the original nonterminal. The second variant is a bit harder to remove.

Removing direct left recursion (and transforming it to right recursion), is a fairly simple process. Illustrated on an example it goes as follows:

$$A \rightarrow A\ \alpha \mid \beta \tag{1}$$

where α and β are strings of nonterminals and terminals, and β does not start with A, can be transformed like the following:

$$A \rightarrow \beta\ A' \tag{2}$$
$$A' \rightarrow \alpha\ A' \mid \epsilon \tag{3}$$

This can be extended to any number of nonterminals:

$$A \rightarrow A\alpha_1|A\alpha_2|\dots|A\alpha_n|\beta_1|\beta_2|\dots|\beta_m \tag{4}$$

where $\beta_1 \dots \beta_n$ do not start with A can be transformed as

$$A \rightarrow \beta_1 A'|\dots|\beta_m A' \tag{5}$$
$$A' \rightarrow \alpha_1 A'|\dots|\alpha_n A'|\epsilon \tag{6}$$

In order to remove indirect left recursion, the nonterminals have to be put in an arbitrary order $A_1 \dots A_n$. Then, you eliminate all rules of the form $A_i \rightarrow A_j\gamma$ where $j < i$ (and γ is some arbitrary string), and replace them with their definition. This does not change the language, but it changes the grammar, and therefore makes the parser more complicated to write and is in general a lot of work.

> Removing left recursion is a bitch, just avoid it from the start if you can. In some cases you might not be able to because right recursion and left associativity don't work too well together.

Left factoring is also very simple: Take every nonterminal starting with the same prefix α:

$$A \rightarrow \alpha\beta|\alpha\gamma \tag{7}$$

and replace them using a new nonterminal

$$A \rightarrow \alpha A' \tag{8}$$

$$A' \rightarrow \beta | \gamma \tag{9}$$

Where α is the longest common substring in the nonterminals.

3.2 LL(1) Parsing

Writing a parser by hand for each grammar sucks. So we want a method of automatically generating parsers for a grammar. One method of achieving this is LL(1) parsing. It is called that because it parses input from left to right, uses a leftmost derivation, and uses one lookahead token.

Generating a LL(1) parser requires a left-factored, right recursive grammar. These parsers use an explicit stack instead of recursive calls, and has only two functions: *match* and *generate*. After initialization, the stack is filled with the stack bottom symbol $\$$ and the start symbol of the grammar. *Match* matches a token in the input stream with the top token of the parse stack, and removes both of them. *Generate* pushes the results of a derivation step in the grammar onto the parse stack in reverse order. LL(1) parsers accept if the input is done, and the stack is empty.

Choosing what to generate is sometimes very hard, so in order to create a transition table for the parser you need first and follow sets, which is annoying as all hell.

4 Bottom Up Parsing

If you read the previous chapter, you know that choosing the right derivation step is hard for top down parsing. This is the major strength of bottom-up parsers, because they can just take incoming tokens until a rule is fulfilled. They are named and differentiated by their derivation policy and and the number of lookahead tokens, but in general they all follow the same procedure. Their two main functions are *shift*, which moves a token (or terminal) to the top of the stack, and *reduce*, which replaces a string α on the stack with a nonterminal A (according to a rule $A \rightarrow \alpha$).

4.1 LR(0) parsing