CSC 488: Building a Compiler

For Beginners

Michael Liut
liutm@mcmaster.ca

January 16, 2017

Contents

1 Introduction **3**
1.1 Software Design . . . 3
1.2 Biting Off More Than You Can Chew . . . 3
1.3 What Does a Compiler Do? What Do I Need To Know? . . . 3

2 The DOs and DON'Ts of Compiler Construction **4**
2.1 DO . . . 4
2.2 DON'T . . . 5

3 Using FLEX and CUP **5**
3.1 FLEX [1] . . . 5
3.1.1 Example . . . 6
3.1.2 Solution . . . 6
3.2 CUP . . . 7

4 Conclusion **7**

1 Introduction

This document has been composed as supplemental material to assist the students of *CSC488H: Compilers and Interpreters* from The University of Toronto with their compiler construction project.

1.1 Software Design

If you recall any of your previous software design and specification courses, you will remember that above all else your primary goal is to ensure that your software works. Once you have a functioning program, you must strive for: elegance, beauty and finesse. Remember, perfection is in the details! Finally, your concern must be efficiency. Optimize you program, but ensure that you never compromise the primary goal; your software must work!

1.2 Biting Off More Than You Can Chew

Over the years I have seen many students take on more than they can handle. Remember, many of you are taking 4-6 courses (some even 7 or 8), building a compiler is a huge task that is not to be taken lightly.
One example I refer to often is the *inverted trapezoid model*. The layers (from top to bottom) are as follows:

1. Software Requirements Specification.
2. High-Level Architecture Design
3. Detailed Design
4. Final Product

The idea behind this model is that at each layer you will shave something off from your initial software requirements specification. As you move into a more detailed design you soon realize the importance of simplicity and will scale down your overall construction. Once you reach your final product (i.e. the compiler) you want to remember what is said in 1.1! If have already forgotten, go back and read it again!

Remember: you must ensure that your input language meets the minimum requires as per the assignment! You cannot "shave" these off.

1.3 What Does a Compiler Do? What Do I Need To Know?

In general, a compiler transforms an input language into a target language. For the purpose of this document I will not be discussing the three major components of a compiler in detail (i.e. the Front End, the Optimizer, and the Back End). Instead, I will be providing a toned down overview to allow you to hone in on key subsections for you to directly apply to your project. I will also imply some suggestions below, I strongly recommend you follow them for your first construction.

Firstly, a compiler is responsible for recognizing the syntax of a language. This means that the sentence(s) of the source program must be checked for validity. What you need to know is that parsing is where the source code, in essence, is converted into an Abstract Syntax Tree (hereinafter known as "AST") and semantically validated with your grammar.
A grammar is a set of rules that explicitly states how to properly form strings in a language's alphabet and ensure they are valid with respect to the language's syntax. What you need to know is that you are best conducting a top-down parsing approach and creating a predictive grammar (thereby being $LL(1)$). Alternatively, you may opt for $LALR(1)$, a superset of LL(*), but I think a predictive

grammar will reduce your work in later sections (i.e. sprints). Also, use of a predictive grammar will remove the risk of memory leaks.

At this point you may be wondering how to transform your code from a high-level (source) language into your desired lower-level (target) language. You may also be wondering how to implement some high-level optimization. What you need to know is that your AST has to be transformed to represent a more efficient computation utilizing the same semantics. Eliminating superfluous local assignments, early calculations of constant expressions and common subexpressions are just a few examples of trivially redundant high-level code that can be optimized at an early stage.

Now you may be asking yourselves, how do I generate the low-level (target) language? The general idea is to utilize the AST by transforming it into a low-level language; commonly x86, MIPS, or ARM. What you need to know is that this is typically left up to the developers and this can widely vary depending on the source language and intended target language. With respect to optimization at a low-level stage (also commonly referred to as peephole optimization), what you need to know is that, the low-level code is scanned for inefficiencies which are then modified and corrected.
Note: both code generation and low-level optimization can be repeated several times.

Let's look at an example:

> If we look at the GNU Compiler Collection (hereinafter known as "GCC"), GCC repeats the code generation and low-level optimization stages many times. It transforms the high-level C code into an intermediary (lower-level) language (which is platform independent). It further optimizes the intermediary code and then transforms this intermediary language into its target (and true low-level) language (e.g. x86, MIPS, ARM, etc...). Optimization occurs several more times, and finally, linking occurs (i.e. the resolution of references to other modules).

2 The DOs and DON'Ts of Compiler Construction

This section will provide you with a brief list of common tips to ease your compiler construction experience.

2.1 DO

1. **Define your language well!**
   Ensure that your language is defined well syntactically and semantically. If it is not, it can turn out to be more work on your part.
2. **Choose a good syntax analyzer!**
   There are many that you can choose from, we will show you how to use FLEX. It is fast, easy to use and open-source.
3. **Choose a good parser!**
   There are many that you can choose from, we will show you how to use CUP; an LALR parser generator. You can use CUP with your predictive grammar. Alternative options to CUP that I have worked with are: ANTLR3/4 (LL(*)), Bison (LALR(1)), and YACC (LALR(1)).
4. **Test your code!**
   Establish a set of test cases and be sure to run them regularly. Remember, regular expressions are your friend and you can use them in your test cases!

5. **Correct errors before moving forward!**
   Running those test cases is one thing, but if they fail you need to fix what is broken prior to moving on. This can be slow, long and tedious; but it is essential!

2.2 DON'T

1. **Code before having your language and design specifications!**
   Skipping to the coding phase without verifying your language and design specifications can turn out to be detrimental.
2. **Get overwhelmed by thinking of the project as a whole from the beginning!**
   This project is broken up into chunks (i.e. sprints) for a reason.
3. **Shoot down others ideas!**
   Give them a chance! Their ideas can prove to be useful!
4. **Make a syntax analyzer!**
   There are many that you can choose from, it is difficult to make one from scratch and quite frankly not the easiest thing to do!
5. **Make a parser!**
   There are many that you can choose from, it is difficult to make one from scratch and quite frankly not the easiest thing to do!

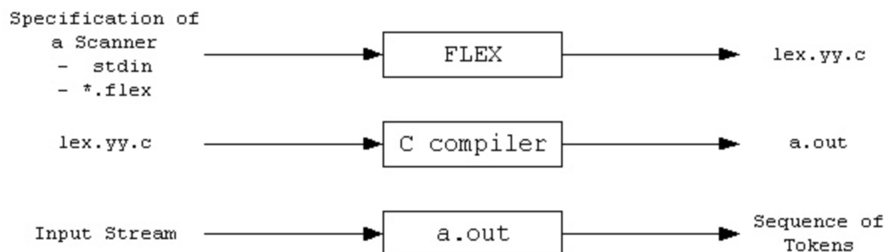
3 Using FLEX and CUP

This section will be used to briefly explain a scanner and parser that you may use for your project.

3.1 FLEX [1]

The scanner performs lexical analysis of your program by reading in the input language as a sequence of characters and recognizing them as tokens. FLEX (Fast LEXical analyzer generator) is used to generate scanners. This is done by identifying the pattern in the text of a certain language. For example, a digit would represent $[0-9]$.

In general, FLEX is utilized in the following manner:



FLEX begins by reading a specification of a scanner (either an input file **.lex* or from standard input) which it uses to generate a C target file as output *lex.yy.c*. This outputted C file is then compiled and linked using the "-lfl" library, which produces an executable file *a.out*. This executable file analyzes its input stream and transforms it into a sequence of tokens.

What you need to know:

- **.lex* uses regular expressions and C code.
- *lex.yy.c* defines a routine *yylex()* that uses the specification to recognize tokens.
- *a.out* is the scanner.

3.1.1 Example

Give a lex/flex input file to generate a scanner that recognizes the following tokens:

1. Java identifier
2. C unsigned integer
3. C literal string

and that 'eats' multi-line comments, where the comment starts with "[|[" and ends with "]|]". The nested multi-line comments are not allowed (similarly as in C/C++/Java). The scanner must treat the comment as white space.

3.1.2 Solution

Given the requirements in 3.1.1., you should have a solution that looks like this:

example.txt

```
%{
        /* Michael Liut */
        /* UofT - CSC488  */
        /* COMMANDS TO EXECUTE:
         *     lex example.lex
         *     gcc -ll lex.yy.c
         *     ./a.out < test
        */
%}

%option noyywrap
%x checkComments

%{
        /* REGEX Variable Initialization */
%}
ws                                 [ \n]|[\n]|[\r\n]|[\t]|[\r]
begComment                 "[|["
endComment                 "]|]"
integers                 [0-9]+
cLiteral                 \"(\\.|[^"])*\"
jIdentifiers         (([a-zA-Z_$])([a-zA-Z0-9_$])*)

%%

%{
        /* Begin Function */
%}
```

```
{begComment} {BEGIN(checkComments);}

%{
        /* Checks for comment end without a start -- Returns -1 */
%}
{endComment} {fprintf(stderr, "Error: Found comment end without a beginning!\n"); return -1;}

%{
        /* Error Checking -- Check for opening comment inside of a comment */
%}
<checkComments>{endComment} {BEGIN(INITIAL);}
<checkComments>{begComment} {fprintf(stderr, "Error: Found [|[ inside another
                                                        [|[ comment!\n"); return -1;}

%{
        /* Removes the comments & comment content */
%}
<checkComments>(.|\n)
<checkComments><<EOF>> {fprintf(stderr, "Error: Found [|[ without closing ]|]!\n"); return -1;}

{integers} {printf("An integer (%s): %d \n", yytext, atoi(yytext));}
{jIdentifiers} {printf("Java identifier: %s \n", yytext);}
{cLiteral} {printf("C Literal: %s \n", yytext);}
{ws} {}
. {printf("ERROR DETECTED %s \n", yytext);}

%%
int main(int argc, char* argv[]){
    yylex();
}
```

3.2 CUP

More to come in time...

4 Conclusion

I hope that you have found this document somewhat useful. Recall, this is document is meant for novice compiler construction. Furthermore, if you have an issues or concerns with your project please do not hesitate to contact me.

I leave you with one quote to think about on your journey in your compiler construction:

> “Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away.”
>
> — Antoine de Saint-Exupéry

References

[1] Lan Gao. Flex tutorial, 2007.