## banner above paper title

Aero 552 Final Report

Alex Schiffer University of Michigan [aschiffe@umich.edu](mailto:aschiffe@umich.edu)

# Abstract

This is the text of the abstract.

## *CCS Concepts* • Software and its engineering Gen- eral programming languages; • Theory of computation

→

→ *Program analysis*

***Keywords*** Regular Expressions, DFA, NFA, Parser, Lexer

# Introduction

The first two stages of compilation are lexing and parsing a source text. Lexing involves grouping the characters of the source text into words (often called tokens) acoording to a set of regular expressions. This stream of tokens is then passed to the parser which groups them into sentences (often in the form of a parse tree) according to a context free grammar. Many lexers and parsers use a deterministic finite automaton (DFA) to process their input. Although the theory behind lexing and parsing is not complicated, generating the DFAs is too tedious to by hand for anything but the simplest languages. Instead, many compilers use lexer and parser generators to generate the DFAs. For my project, I implemented a lexer generator and a parse table generator.

# Theory

## Deterministic Finite Automata

Efficient lexing and parsing both depend on deterministic

* + - *F* ⊂ *Q* is the set of accepting states of the DFA

The DFA works by its current state *si* and current input symbol *c*. The DFA is initially in its initial state. To select the next state, it calls the state transition function on *si* and *c*, *δ*(*si, c*). It repeats this process until either there are no more input symbols or there is no valid transition. If there

are no more input symbols, the DFA accepts the input and optionally performs some action according to the input state. If the DFA is not in an accepting state or there is no valid transition, the DFA reports an error. The set of input symbols that can be recognized by a DFA is called its language.

## Lexing

Lexing is the first phase of compilation; it involves splitting an input stream into tokens based on a set of regular expres- sions.

## Regular Expressions

A regular expression is a compact way to represent the language represented by a DFA. Regular expressions are constructed from a few simple operations.

|  |  |
| --- | --- |
| Regular Expression  a | Language  L(a) = {a} |
| *ε* | L(*ε*) = { } |
| ab | L(ab) = {*αβ*|*α* ∈ *L*(*a*)*, β* ∈ *L*(*b*)} |
| *a*|*b* | L(*a*|*b*) = *L*(*a*|*b*) = *L*(*a*) ∪ *L*(*b*) |
| a\* | L(a\*) = ∪*i≥*0*L*(*a*)*i* |

finite automata (DFA). A DFA is a 5-tuple, where

* + - * *Q* is a finite collection of states

{*Q*,

Σ, *δ*, *s*, *F* }

The lexer uses a set of regular expressions to construct a

* + - * Σ is a finite collection of symbols that the DFA can recognize called its alphabet
      * *δ* : *Q* ×Σ → *Q* is the state transition function of the DFA
      * *s* ⊂ *Q* is the initial state of the DFA

DFA that can be used to split an input string into tokens. The lexer uses the following algorithm.

**Listing 1.** Lexer Algorithm

s i *<* s 0

−

w h i l e ( more i n p u t c h a r a c t e r s ) c *<* n e x t t o k e n

−

s i = d e l t a ( s i , c ) i f ( s i == e r r o r )

p r i n t e r r o r

e l s e

advance

[Copyright notice will appear here once ’preprint’ option is removed.]

end i f

end w h i l e

i f ( s i == a c c e p t ) p r i n t a c c e p t

merged *<* marged ( c , n1 , n2 ) n f a s t a c k . push ( merged )

r e t u r n n f a s t a c k . t o p ( )

−

e l s e

p r i n t e r r o r

## 2.4 DFA Construction

The lexer uses Thompson’s construction and the powerset construction to convert a set of regular expressions to a DFA.

## Thompsons’s Construction and Non-deterministic Finite Automata

The first step in creating a DFA from a set of regular ex- pressions is to create a non-deterministic finite automaton (NFA). Like a DFA, an NFA s a 5-tuple *Q*, Σ, *δ*, *s*, *F* where

{ }

* + - *Q* is a finite collection of states
    - Σ is a finite collection of symbols that the DFA can recognize called its alphabet
    - *δ* : *Q* ×Σ → *Q* is the state transition function of the DFA
    - *s* ⊂ *Q* is the initial state of the DFA
    - *F* ⊂ *Q* is the set of accepting states of the DFA

However, unlike a DFA, an NFA is allowed to have non- deterministic transitions. This means that it is possible to have epsilon transitions, translations where the input char- acter is empty, and have transitions where the same input character can lead to different states. It is hard to simulate an NFA, but it is easier to construct an NFA from a regular expression than a DFA. An efficient construction to do this is called Thompson’s construction.

Thompson’s construction works by building an NFA from pieces of the regular expression. The first step is to convert a regular expression from infix notation to postfix notation. Then, you create a NFA for each character in the expression and merge NFAs together based on the operators in the NFA. When the construction process is over, there should only be one NFA left. An algorithm for Thompson’s construction is shown below.

**Listing 2.** Thompson’s Construction n f a s t a c k *<*

w h i l e ( n o t a t end of e x p r e s s i o n ) c = n e x t c h a r a c t e r

— {}

i f ( ! i s o p e r a t o r ( c ) )

n *<* c r e a t e n f a ( c ) n f a s t a c k . push ( n )

−

e l s e i f ( c i s \*)

n1 *<*− n f a s t a c k . pop ( ) merged *<* merge ( \* , n1 ) n f a s t a c k . push ( merged )

−

e l s e

n1 *<*− n f a s t a c k . pop ( ) n2 *<*− n f a s t a c k . pop ( )

Once the regular expression is turned into an NFA, the NFA can be converted to a DFA; this process is called the power- set construction. Intuitively, given a state *s* in the NFA, the powerset construction condenses the set of all NFA states that can be reached from *s* upon seeing a given input charac- ter *c* and *ε* transitions. It is essentially performing a breadth- first search in the NFA, where the successor function is the states that can be reached from the current state upon see- ing a given input character and *ε* transitions and the goal is finding all such states. The powerset construction relies on the epsilon closure function which returns the set of states that can be reached from the current state using only *ε* tran- sitions.

**Listing 3.** Powerset Construction D0 *<*− e p s i l o n c l o s r e ( s 0 )

D *<*− {} / / DFA S t a t e s

— {}

T *<*

w o r k l i s t *<*

−{}

w h i l e ( w o r k l i s t i s n o t empty )

{ N i *<* w o r k l i s t . pop ( ) t *<*

— {}

−

f o r ( each c h a r i n a l p h a b e t )

{ N j *<* d e l t a ( N i , c h a r )

−

t = e p s i l o n c l o s u r e ( N j )

T ( N i , c h a r ) = t i f ( ! t i n D)

D. i n s e r t ( t )

w o r k l i s t . add ( t )

}

}

Each set of states in D will become a state in the DFA. Using D and T, it is possible to construct the state transition table for the DFA. This DFA can then be used in the lexer algorithm presented to lex a source text.

## Context-Free Grammars

Although regular expressions can describe the tokens of lan- guages, they are not powerful enough to describe the syntax of languages. The syntax of languages can be described us- ing context-free grammars. A context-free grammar is a set of rules that describes how to form sentences. Each rule takes the form

N → *α*N N → *α*

The first rule indicates that N can derive *α* followed by an- other N. *α* is called a terminal; it represents a word in the

language and cannot appear on the left-hand side of the rule. N is a non-terminal and represents a set of strings that can be derived from the grammar. All grammars must have a start symbol that represents the set of all possible strings that can be derived from the grammar.

To derive a sentence from a grammar, start with the start rule. In each rule, replace each non-terminal with one of the rules it appears on the left-hand side of. Repeat this process until there are no more non-terminals. At each step, it is pos- sible to expand the leftmost non-terminal first resulting in a leftmost derivation or to expand the rightmost non-terminal first resulting in a rightmost derivation. Although the left- most and rightmost derivations will apply the grammar rules in different orders, they will produce the same sentence.

## LALR(1) parsing

The input to a parser is a stream of tokens produced by a lexer. The parser then determines if the tokens form a valid sentence based on a context-free grammar by forming the tokens into a parse tree. The leaves of the parse tree are the tokens in the input stream and the structure of the parse tree encodes the structure of the sentence. There are two broad categories of parsers: top-down parsers that form the tree from the root working to the leaves and bottom-up parsers that form the tree working from the roots up. My parser generator creates a table for a particular type of bottom-up parser called an LALR(1) parser.

The LALR(1) parser belongs to a class of bottom-up parsers called shift-reduce parsers. The bottom-up parser starts by forming partially constructed trees from the input tokens. These initial trees will form the leaves of the final parse tree. When the parser sees a group of partially constructed trees that forms the right-hand side of a rule in the grammar, it creates a new node representing the left-hand side of the rule. It then makes the group of trees the child of the new node. The parser continues this process until it combines all the partially constructed trees into the goal symbol, or the parser is unable to merge a group of trees. In the first case, the parser accepts the input. In the latter, the parser rejects the input and should report and error.

At the heart of a shift-reduce parser is a DFA and a stack. As the parser reads input tokens, one of two things can happen. If the token completes a rule in the context free grammar, the parser can reduce the stack. It pops off all the tokens in the stack contained in that rule and pushes a symbol repre- senting the rule in the stack. This is equivalent to adding a new node to the parse tree. If the token does not complete the rule, the parser shifts the input token onto the stack and goes to the state indicated by the DFA.

There are two parts to the parse table: the goto table and the action table. The goto table is used immediately after the

parser reduces. When the parser reduces, it looks up state to go to based on the parser’s current state and the rule it used to reduce the stack. The action table is used to determine what action to perform when it reads an input token. Based on the current state and the input token, the action table tells the parser whether to shift the token onto the stack or reduce with a particular rule. In literature, this is often indicated as ”rN” and ”sN.” For ease of implementation, I chose to rep- resent shift actions as positive numbers and reduce actions as negative numbers.

Unfortunately, it is more difficult to create the action and goto tables from the context-free grammar than it is to cre- ate the lexer DFA from a set of regular expressions. There is no equivalent of Thompson’s construction for context-free grammars. The first step is to augment the context-free gram- mar with a pointer indicating where in the rule the parser currently is. Then, the parser generator groups the rules into item sets. Each item set will eventually represent one state of the parser. The formation of an item set begins with a rule called the kernel. If the kernel’s pointer is pointing to a non-terminal, all rules beginning with that non-terminal is repeated. For each of the rules added, if any of those rules’ pointers is pointing to a non-terminal, all rules beginning with that non-terminal are added to the set. This process continues until there are no more rules to be added. The algorithm for this is shown in the listing below.

**Listing 4.** Create Item Set

I *<* k e r n e l

— { }

w o r k l i s t *<* k e r n e l

— { }

w h i l e ( w o r k l i s t n o t empty ) r u l e *<* w o r k l i s t . pop ( )

−

s *<* symbol a t r u l e ’ s p o i n t e r i f ( s i s non t e r m i n a l )

−

−

f o r ( each r u l e i n grammar ) i f ( ! r u l e i n I )

I . add ( r u l e )

w o r k l i s t . add ( r u l e )

One the first item set is created, the next are item sets are created by ”giving” the first item set each input in the gram- mar’s set of inputs (the union of the terminals and non- terminals of the grammar). For each rule whose pointer is pointing to that input, the pointer is advanced, and that rule is used as the kernel of a new item set. Item sets must be unique, so if that kernel leads to an item set that already exists, the item sets are merged. Once all inputs have been provided to the first item set, the process is repeated with the other item sets until there are no new item sets created.

The item sets are then re-organized into a translation table. The table indicates what item set (state) to go to given the current item set (state) and current input symbol; this table will become the basis for the action and goto tables.

The goto table is simply formed from the columns of the translation table containing non-terminals. The action table requires the creation of two new sets: FIRST sets and FOL- LOW sets. The FIRST set of a symbol represents the set of terminals that can begin expressions derived from that sym- bol. The FOLLOW set of a non-terminal represents the set of terminals that can follow that non-terminal in a derivation.

The FIRST set of a symbol can be generated from 5 rules.

1. FIRST(*<*) = ∅
2. FIRST(terminal) = *terminal*
3. FIRST(*αβ*) = FIRST(*α*) if not NULLABLE(*α*)
4. FIRST(*αβ*) = FIRST(*α*) ∪ FIRST(*β*) if NULLABLE(*alpha*)
5. FIRST(N) = FIRST(*α*1) ∪ *. . .* ∪ FIRST(*αn*) where N →

*α*1 ∨ . . . ∨ N → *αn*

where N is a non-terminal. A symbol is NULLABLE if there exists some derivation *α ε*. The FIRST sets are calculated by using fixed-point iteration. The FIRST set for every sym- bol is initially empty. Then rules 1-5 are applied to each, using the assumption that each FIRST set is initially empty where applicable. Rules 1-5 are then repeatedly applied until none of the first sets are changing.

→

The FOLLOW set of a non-terminal can be generated creat- ing a system of constraints and then solving that system of constraints. There are three constraints used in the genera- tion of FOLLOW sets.

1. $ ⊆ *FOLLOW* (*S*)
2. if *N Mαβ* where *α* is NULLABLE, *FIRST* (*β*)

→ ⊆

*FOLLOW* (*M* )

1. if *N Mα* where *α* is NULLABLE, *FOLLOW* (*N* )

→ ⊆

*FOLLOW* (*M* )

The FOLLOW sets for all non-terminals are initialized using constraints 1 and 2. Then constraint 3 is repeatedly applied until the FOLLOW sets are no longer changing.

Once the FOLLOW sets have been generated, the action table can be generated. First, all the columns containing ter- minals from the translation table are copied to the action table. They are re-labeled to indicate that they are shift ac- tions. Then, for each rule, figure out what item set that rule ends in. To create the reduce actions, the parser generator must figure out what state the parser will be in when it has read all the symbols corresponding to a rule and what ter- minals can follow the non-terminal the rule is reduced to. It does this by finding the item set containing the rule with the pointer at the end of the rule; that is the state the parser will be in when it has read all of the symbols correspond- ing to that rule. It then uses the FOLLOW set of the rule’s left-hand side to determine what terminals may follow the

rule. With these two pieces of information, the parser can generate the reduce actions in the action table. Unlike the shift actions, value placed in the action table indicates which rule to reduce by rather than which state to go to.

# 3. Code

My program consists of two parts: a lexer generator and a parser table generator. I wanted to write a full parser gen- erator; however, I did not have time. The input to the lexer generator is a text file containing regular expressions and the output of the program is a C++ program that represents a lexer that lexes as sequence according to the specified reg- ular expressions. I am sure there are better ways to create a C++ program within a C++ program; however, for the sake of simplicity, I chose to create a std::string that contains the header and .cpp files of the lexer. I simply wrote this file to a string. The input to the parser generator is a text file contain- ing a grammar, and the output is two DFAs that represent the LALR(1) parse table. To go from the parser tables to the parser generator; the parser tables could be embedded into another C++ program like in the lexer generator.

Overall, this project was probably one of the most coding projects I have ever done. I have studied compiler design in- dependently for several years, but I had never implemented a lexer or parser generator by hand. I assumed that my knowl- edge of the theory behind lexing and parsing would translate easily to code. I was very wrong. Although the theory be- hind lexing and parsing is not complicated, translating this theory into clean data structures and efficient algorithms is very difficult. There are many edge cases and difficulties that the programmer needs to consider that are not immediately apparent when creating lexers parsers by hand. For exam- ple, when creating FIRST and FOLLOW sets, it is necessary consider how all of the sets simultaneously. This is easy to do by hand but implementing an algorithm that can do this is very difficult. It is very easy to create an infinite loop.

The most difficult part of the project is translating that math- ematical definition of automata and the algorithms used in constructing them into data structures that can be easily tra- versed. Especially for the construction of the parse tables, it became important to balance the need to have a data struc- ture that supported random access with the need to have the ability to easily determine if an element is in the data structure. For algorithms that used fixed point iteration, such as the algorithms for generating first and follow sets in the parser, it was more important to have the ability to easily determinate if an element is in the data structure. When- ever a data structure holds elements that will become states in an automaton, it was important to have random access. Because C++ has different interfaces for working with these different data structures, making this decision early was very important as it is difficult to change it once the algorithms

are implemented. The C++ standard library was very useful in implementing the generators; however, the standard tem- plate library in particular requires that code be in a certain format. It is easy to forget that the STL requires this for- mat when the format is not required when performing the algorithms by hand (for example, you need to write custom comparison functions and hash functions). The STL is no- torious for its difficult to read error messages, and it did not disappoint for this project. I spent many hours deciphering error messages that were hundreds of lines long.

Writing the lexer generator presented some unique difficul- ties. Because the lexer is an output of the C++ program, any errors in the lexer cannot be fixed directly. Instead, I need to find the code in the lexer generator responsible for gen- erating that part of the lexer and fix that code accordingly. This means it is necessary to think about C++ code as a std::string; it is easy to forget that std::string requires escape characters for common characters in code such as newlines, ”, and ’. I also had to think about interweaving values stored in the generated DFA with hard coded string. This makes the debugging process more tedious as every change that needs to be made to the lexer requires altering the lexer generator, recompiling, and re-running the lexer generator.

Another important decision was writing data structures to represent the automatons. My goal was to create data struc- tures that were minimal but still contained enough informa- tion. Eventually I decided on a design that only kept stored the transition function (as a table) and the accepting states for the DFA, and the transition function, accepting states, and alphabet for the NFA. The transition function is repre- sented as a vector of maps; the list of states is maintained implicitly as the indices of the vector. I also wanted the same DFA data structure to be used for both the lexer and the parser tables.

One more difficult decision was determining the overall struc- ture of the parser and lexer generators. Each contains many phases that all need to interact with each other. The lexer generator needed to go from regular expression to DFA; this process could be split naturally split into converting the reg- ular expression from infix to postfix, creating an NFA from the regular expression, creating the DFA from the NFA, and finally creating a lexer from the DFA. Additionally, each of the steps decomposes naturally into smaller subproblems. This made it easier to create the interfaces for each of the steps. Additionally, the process is entirely linear; the result of one phase is needed only by the following phase. Converting the regular expressions from infix to postfix and converting regular expressions to NFAs using Thompson’s construction translate especially well to a computer program. Both are variations on the well-known problem of evaluating arith- metic expressions; it is easy to pick the data structures.

The generation process is made both easier and more diffi- cult by the fact that many algorithms are similar in structure, require similar in structures, and produce similar outputs. This simplifies the code because it much of the code can be reused. However, this requires writing code that can be reused. When I designed algorithm skeletons and data struc- tures, I had to consider several things. Could another func- tion reuse this code? If so, will the code be reused be almost verbatim or will need to be tweaked slightly? In addition, if code is reused, it needs to be tested very thoroughly first. Any bugs present in the original contexts will cascade to other contexts, which could make the program very difficult debug.

Something I am very proud of is the unit test framework I wrote to help make testing easier. Writing unit tests is very important but also very tedious. The unit test framework au- tomated all of the tedious aspects of writing tests allowing me to focus on actually creating test cases. However, it had an unintended limitation. The unit test framework is very useful for comparing the lexer generator/parser generator solutions against an expected solution; however, it is only able to perform exact matches. This is useful for the regular expression parser and NFA generator. However, there are many possible DFAs for a given NFA and regular expres- sion, and the unit test framework cannot tell if two solutions are identical. Additionally, each of the phases of the parser table generation has many possible solutions. If I had more time, I would write a more flexible unit test case.

A second difficult part of testing is that is how much ”book- keeping” is needed to create DFAs and parse tables for any- thing but the simplest regular expressions and context-free grammars. Parse tables especially are difficult to create by hand. This means that tests need to be done on small gram- mars that contain all of the edge cases that could be found in larger grammars. Using Thompson’s construction, NFAs are assembled in pieces corresponding to sections of regular expressions. It is easy to write regular expressions that con- tain the edge cases that can be found in any arbitrary regular expression fed through Thompson’s construction. However, DFA and parse table generation is more complicated. For DFA generation, I tried to create regular expressions that would test for edge cases, but it is possible that larger gram- mars contain cases that I missed. Constructing parse tables by hand is even more difficult, so it is very difficult to test the parser generator. While I tested it as well as I could, there may be classes of grammars that are valid that it can- not properly handle.

One thing I wish I could have done differently was create a different interface for the parser generator. The parser gen- erator has many phases, and I wanted to be able to test each

one individually. Initially, I intended to structure the parser generator like I did the lexer generator; however, many of the phases need to access a shared state. This led me to make a parser generator data structure. Since this data struc- ture was made from necessity rather than by design, it is not as clean as I would like it to be. If I could start over, I would have started with a parser generator data structure. Hopefully, doing so would have allowed me to implement a cleaner design.

This project was very difficult, but it was a great learning experience. First, I learned that you should always allocate more time than you think is necessary for the project. One of the biggest challenges of this project was the time con- straint, and I unfortunately fell a little short of what I had hoped to choose. If I could start over, I would have overes- timated how long the project would take and narrowed my scope accordingly. Second, I learned how to use build sys- tems to keep a large project organized. By using CMake, I could create a logical structure for my project. Trying to cre- ate the same structure would require Makefiles or command line commands that are very difficult and tedious to write by hand. It also allowed me to easily change the structure of my project when required. Third, this project improved my ability to translate complicated algorithms into code.

# References

Cernera, G., ’Converting Regular Expressions to Post- fix Notation with the Shunting-Yard Algorithm,’ Avail-

able: https://gregorycernera.medium.com/converting-regular- expressions-to-postfix-notation-with-the-shunting-yard- algorithm-63d22ea1cf88.

Cernera, G.,’Visualizing Thompson’s Construction,’ Available: https://medium.com/swlh/visualizing-thompsons-construction- algorithm-for-nfas-step-by-step-f92ef378581b.

Cooper, K. D. and Torczon, L., ”Scanning,” *Engineering a Compiler*, Elsevier Science, California, 2004, pp.27 - 72.

Holub, A., I., ”Top Down Parsing,” *Compiler Design in C*, Prentice- Hall, Inc., New Jersey, 1990, pp.195 - 336

Jackson, S., ’LALR(1) Parsing’ Available: https://web.cs.dal.ca/ sjackson/lalr1.html.

Morgesen, T.A., ”Syntax Analysis,” *Basics of Compiler Design*, University of Copenhagen, Copenhagen, 2000, pp.53-111.