Background Report: Parse Master 9000:

Parser and Define-Type Generation for EBNF Language Specifications

Etienne Le Bouder
University of British Columbia
Vancouver, British Columbia, Canada
etienne.lebouder@alumni.ubc.ca

Connor MacCuspie
University of British Columbia
Vancouver, British Columbia, Canada
c.maccuspie@alumni.ubc.ca

Haosheng Zheng
University of British Columbia
Vancouver, British Columbia, Canada
logan.zheng@alumni.ubc.ca

Dongan Liu
University of British Columbia
Vancouver, British Columbia, Canada
dongan.liu@alumni.ubc.ca

ABSTRACT

An overview of the construction of an LALR parser-generator for EBNF language definitions, done in Dr.Racket after analysis of different parsing approaches. To aid with parsing the broad EBNF grammars, an additional analysis of potential restrictions and the development of a user-input driven type definition will be added to the original research on parsing approaches, benefits, and the role they play in compilers.

KEYWORDS

Parser, Lexical Analysis, Interpreter, Syntax Tree, Syntax, Functional Languages

1 OVERVIEW

For every input, there is an equal output. While not quite one of the natural laws, it holds in much of computer science. When a program is given some input, there must be some related output from the program as a result. The most important element of this quasi-state machine representation of a computer is regarding how input is handled.

Moreover, how does a program represent the input internally? Without knowing what the input is, there is no way to assign meaning and determine what the input means. Discerning what exactly a piece of code is asking of a program is the job of the first step at compile time: parsing.

Input rejection and acceptance defines a language [2] and this is the key role of the parser. Whatever constitutes a valid input for a language must be accepted, and every other input rejected. With this key goal in mind, there are different ways in which it can be accomplished.

Parsers can be written by hand to handle their language-specific input and grammar, or they can be generated. The latter technique has been a particular favorite for many reasons, namely verification that there are no grammatical ambiguities or conflicts, accepting only valid inputs as defined by the given grammar, and the ability to recover from various errors and continue parsing [2].

How parsers consume and break up the input string is left to the implementation and type of parser generator. Some of these approaches include LALR, LL, and LR(k), amongst others. These vary mostly on how they construct the internal representation of the given input, and how they traverse the input string. The former involves construction of the syntax, or parse tree, and whether this is done from the bottom up, or the top down. Traversing the input string can be done in various ways, most often by looking ahead or looking behind.

We explore the role of parsing, parser technique and theory, and the compilation pipeline with reference to the construction of a program to generate parsers. Using the Racket define type construct and a user given EBNF, what is outlined in this report will be applied to generate a substantial and robust program to generate parsers from the given information.

2 PURPOSE OF PARSING

The best way to understand parsing is to examine how people naturally do it. Compilation of a program is no different from a person reading and comprehending a sentence. The input string (characters) is read and broken up into meaningful components (words and grammar). If the string is malformed due to some sort of spelling or grammatical error, humans can often recover from this error and continue on with reading the input string. Once reading is complete, this input string is evaluated until understood. People parse strings of letters into words, then evaluate the words and punctuation into meaning.

Continuing with this example, to comprehend the meaning of a sentence, an essential step is to break down the sentence into different components. In human language, we tend to split the sentence into objects, actions, descriptions and learn about their functions in the sentence. In computer language, the same process, defined as parsing, is often applied by the compiler to dissect various parts of an expression into values. The value of parsing can't be underestimated since all languages have their own set of grammar rules [6].

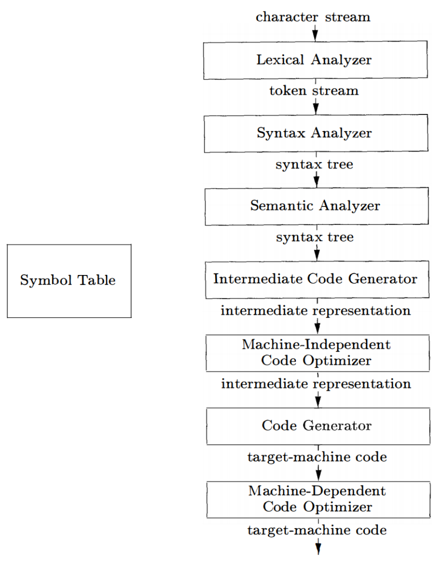


Figure 1: Phases of Compilation

3 COMPILER COMPONENTS: AN OVERVIEW

3.1 Compiler Process

The compilation process needs to go through a sequence of seven phases as shown in Figure 1, where each of the phases will transform the representation of the source code into object code, which in turn is assembled into code for the specific language. A symbol table is also generated from the source code to be used by all phases of the compiler. [1]

3.2 Lexical Analysis

Lexical analysis is the first step of a compiler. It is the preparation stage before the syntactic analysis. Lexical analysis is often called 'scanning' or 'tokenization', and it's the process of reading the stream of characters within source code, and grouping the characters into meaningful sequences called 'lexemes'. The parser will search for these lexemes while it reads the input it is given [1].

In lexical analysis, a lexeme is defined as "a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as instance of that token"[1]. The token is often shown in the form of <token-name, attribute-value> where attribute-value is optional, and it is the atomic unit of a language [4]. Moreover, the token name is an abstract symbol representing a kind of lexical unit. There are several kinds of tokens which are used in most programming languages such as if, else, comparison, id, number and literal classes [5]. The token name is important because it is the input symbol that the parser processes in the next stage. After the tokenization is done, the compiler will then move on to 'syntax analysis' which is commonly known as parsing.

3.3 Parsing

Parsing covers the syntax analysis phase of the compiler. Given the tokens generated from lexical analysis, (in our project, the given EBNF and define type can be used in place of this) the parser recognizes tokens and grammar rules in the input stream. If the stream contains valid inputs, the parser creates its representations of them. If invalid, errors are thrown and parsing either handles them and continues, or exits and compilation fails. Parsing should never allow invalid inputs through the rest of the compilation process.

3.4 Parser Generation

Intermediate code generation is covered by the parser generator. The code that is created is equivalent to the parsers written in Dr.Racket during 311. They are capable of turning the concrete syntax given by a programmer into a compiler-friendly representation of abstract syntax. This process is hidden in the compiler, as opposed to being explicitly defined and given as output. The final milestone for this project will be a concrete program that represents what a compiler using our code would generate as an intermediate step before interpretation and compilation completes.

3.5 Continuation of Compilation

Once parsed, the internal representation of the language is moved through the rest of the compilation process,which is beyond the scope of this project. The constraints placed on parsing, namely that all valid inputs must be accepted and all invalid inputs rejected, are due to the downstream process. Execution time is expensive, so catching errors by scanning in the parser is much easier than trying to catch malformed syntax later on in compile time. Errors and issues that make it beyond parsing can be handled alongside the rest of compilation, with the assumption that syntax and most of the grammar is correct. This division of labor is very advantageous.

4 GRAMMAR

Grammar is a tool for the purpose of describing and analyzing languages. It is a set of rules by which valid sentences in a language are constructed. In compiler design, Context-free Grammar is used to support parsers [7].

4.1 Formal Grammars

4.1.1 Context-Free Grammars: Parsers use token stream and a tool called CFG (Context-free Grammar), a particular class of relevant to programming languages. This context-free grammar is the superset of regular Grammar, and has a larger scope than the regular grammar. It mostly defines the allowed syntax of programming languages [4].

There are four parts in a CFG:

(1) Non-terminals: Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings which help define the language generated by the grammar.
(2) Terminals: Terminals are the basic symbols from which strings are formed.

(3) Productions: The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each of the productions has 3 parts:
   - A non-terminal called the head or left side,.
   - The symbol '$\longrightarrow$' or '::='
   - A body with zero or more terminals and non-terminals. The components of the body describe one way in which strings of the non-terminal at the head can be constructed.

(4) Start Symbol: A start symbol is a special non-terminal, and it is the start of a production.

By using the Context-free Grammar, the syntax analyzer or parser can generate the parse tree for the next phase.

4.1.2 Recursion and Ambiguity: A grammar is considered ambiguous if it's capable of generating more than one parse tree (left or right derivation) for at least one string [4]. While some languages can accept both ambiguous and unambiguous grammar, many don't accept ambiguity. The issue with ambiguous grammar is that one cannot uniquely identify a parse tree to represent the sentence. Ambiguous grammar can often be rewritten to eliminate its ambiguity [1].

In terms of context-free grammar, a non-terminal is left-recursive if the leftmost symbol in one of its productions is itself (in the case of direct left recursion) or can be made itself by some sequence of substitutions (in the case of indirect left recursion) [9]. For example, A $\longrightarrow$Aa is a left most recursive production where the leftmost non-terminal is the same as the head of the production. The left recursion can cause the parser to loop forever, but it can be fixed in multiple ways including rewriting the offending production to A $\longrightarrow$Aa| B. The A will finally be replaced be B after producing a sequence of a's (zero or more) [1].

4.2 Grammar Specifications

Terminal definitions and rule expressions for context-free grammar are critical to parser design. These are the components that make automated parsing feasible and help introduce grammar constraints to the parsing process. One of the notation languages that explores these components is EBNF.

4.2.1 EBNF. As an extension to the Backus Naur Form notation technique, EBNF has been widely recognized as the standard notation to express context-free grammar rules in a formal language. The language consists of a terminal set T, non-terminal disjoint set N from T, a start symbol S from N, and a set of rule expressions A for at most one of each non-terminal from N. [8]. For the terminals, the strings that define them must be enclosed in quotation marks, and for the rule expressions, the right-hand sides are described in regular expressions over terminals T and non-terminals N.

4.2.2 Ambiguity Revisited. Despite its expansive representation, EBNF still carries over some of the restrictions introduced in context-free grammar, particularly grammar ambiguity and recursion mentioned above.

Here's an EBNF example on basic arithmetic operations:

```
Expression = Operand, {[Operator, Operand]};
Operand  = "0"|"1"|"2"|"3"|"4"|"5"|"6"
           |"7"|"8"|"9";
Operator = "+"|"-"|"*"|"/";
```

Even though this specification expresses the operations well, this doesn't resolve differences in priority order for operators in an arithmetic expression since all of them share equal representation. (e.g. 1 + 2 * 3 can be interpreted as (1 + 2) * 3 or 1 + (2 * 3))

4.2.3 Left Recursion. One of the advantages of EBNF over its original BNF form is that the language can define repetitions and recursions without intermediate rules [8]. However, this brings context-free grammar's left recursion constraints into light. Usually an EBNF rule with left recursion is expressed as A $\longrightarrow\alpha$A | β, and one of the resolving methods is to introduce intermediate non-terminal values and replace the original rule with two expressions: A $\longrightarrow\beta$A', A' $\longrightarrow\alpha$A' | ϵ, assuming $\alpha \neq \epsilon$, where A and A' is non-terminal and rest of the symbols are terminal.

5 FROM 311'S LL TO LR

Throughout the course of CPSC 311, we have iterated through the development of simple parsers for the various languages we have explored. However, these parsers have been limited to those using a top-down, syntax-matching approach to transforming concrete syntax into interpreter-compatible abstract syntax.

To better illustrate this, let us briefly consider the following simple EBNF for an arithmetic language.

```
<AE> ::= <num>
       | { + <AE> <AE> }
       | { - <AE> <AE> }
       | { * <AE> <AE> }
```

In a 311-style parser, the parser would take in a (presumably valid) program in concrete syntax, and beginning from the start symbol, would attempt to recursively match the concrete syntax to the right-hand-side productions in the EBNF until no more productions can be made. (Will add more here to describe the top-down parser).

This form of recursive top-down parsing is known as the LL parsing strategy. While LL parsers are typically simpler and easier to build manually, they are often abandoned in favor of bottom-up (LR) strategies that will be discussed shortly. While they are more complex, LR systems are considered to be more powerful as they can be applied to a broader set of grammars and are more robust against ambiguities and left recursion. Indeed, common parser generators including Yacc, Bison, and ANTLR all adhere to various bottom-up parsing strategies.

6 LR PARSING PROCESS: THE FIRST BOTTOM UP STRATEGY

Unlike LL parsing schemes, a bottom-up parser begin with a string of terminals, and iteratively assembles substrings until the right side of some production has been found. It then reduces the substring to the left non-terminal side of the production, which may be then further assembled into another substring, reduced, and so forth until the beginning starting symbol of the grammar has been reached. At this point, the concrete syntax string has been verified to belong to the grammar in question, and parsing is complete.

The most powerful and common form of parser is the shift reduce parser, which will be explored at length in the following sections. It is primarily composed of a stack, which keeps track of the parsing progress, and a parsing table to determine the next action taken by the parser. As the input stream of terminal tokens is processed, tokens are pushed onto the stack until a right-side production, or a handle, is found and reduced. For instance, given a production A →wq, if the top element of the stack is w and the next 2 tokens of the input stream are q+, we may shift q onto the stack and reduce such that the top two stack elements w and q are replaced with A, and + is the next token in the input stream. If the entirety of the input stream has been processed and the only remaining element in the stack is the start symbol, the program has been successfully parsed and and "accept" state will be attained. If not, an error will be thrown.

In practice, however, there is far more sophistication to the function of an LR parser. The reduction rules for handles depends upon the context of the sentential form that is being considered, so tokens are typically replaced with more abstract states before being shifted onto the stack. These states describe the current left context of the parse, and combined with a look-ahead into the input stream determine whether we have a handle to reduce or whether more states must be shifted onto the stack.

In addition to the stack, an LR parser uses an Action table and a GoTo table. Given state S on top of the stack and next input token A, Action[S, A] tells us whether to reduce, shift, accept, or throw an error. Should we reduce S to non-terminal X, GoTo[S, X] tells us what state to shift onto the stack next. We will discuss this in greater detail in subsequent sections.
There are a multitude of LR parsing paradigms of varying efficacy and power. The main categories are LR(k), SLR, and LALR. LR(k) denotes an LR system with k tokens of look-ahead: thus LR(0) is the simplest and the least powerful form of LR(k) as it cannot gather any context from the sentential forms to help decide when to reduce and when to shift. As such we will gloss over it and instead discuss SLR and canonical LR(1) parsers, until enough background has been established to cover the improvements that make LALR such a dominant parsing paradigm today.

6.1 Closure, Configuring Sets and Successors

Before we can continue, we must introduce the idea of configurations. In the simplest case, suppose we have the some production A →XYZ. We will have 4 possible configurations of this production, as follows:
$A \rightarrow \bullet XYZ$
$A \rightarrow X \bullet YZ$
$A \rightarrow XY \bullet Z$
$A \rightarrow XYZ\bullet$
The black dot signifies how far we have gotten into parsing this production. Everything to the left of the dot has been shifted onto the parse stack: thus when we have reached the final case, we have a handle we can reduce. However, suppose we are at an intermediate case such as $A \rightarrow X \bullet YZ$, where Y has the production Y →U | w, and U →z. At this point, the following productions are equivalent in state:
$A \rightarrow X \bullet YZ$
$Y \rightarrow U$
$Y \rightarrow w$
$U \rightarrow z$
Since these correspond to outwardly different sentential forms for the same stack state, we put all these equivalent items into a single set and call it the configurating set. Every entry in the Action/GoTo tables corresponds to a single configurating set. Likewise, each configurating set describes the paths that are being concurrently explored by the parser at some state. Of course, this means that the process of generating parse tables involves generating these configurating sets for the grammar in question. The action of generating a configurating set is called a closure.
Another definition we must make is that of the successor function. The successor defines the transitions between configurating sets: that is, given a configurating set C and a symbol X, we calculate the successor set C' = successor(C, X). C' then describes the configurating set we move to when we encounter X in state C. Fortunately, the successor is easily computed; we merely move one token forwards in our parse and compute the next resulting configurating set.
The following steps outline an algorithm for generating configurating sets [7].

1. A →• u is in the configurating set
2. If u begins with a terminal, we are done with this production
3. If u begins with a nonterminal B, add all productions with B on the left side, with the dot at the start of the right side: B →• v
4. Repeat steps 2 and 3 for any productions added in step 3. Continue until you reach a fixed point.

6.2 Action/Go-to Tables

With the configurating sets defined in 6.1, we may take a step further and define the rules of building a parse table. We can implement a canonical LR(1) style parser as presented in [7].

(1) Construct F = {I0, I1, ... In}, the collection of configurating sets for the augmented grammar G' (augmented by adding the special production S' →S).

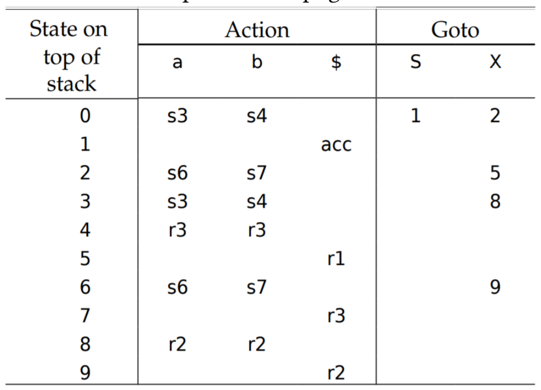
(2) State i is determined from Ii. The parsing actions for the state are determined as follows:

  (a) If [A →u•, a] is in Ii then set Action[i,a] to reduce A →u (A is not S').

  (b) If [S' →S•, $] is in Ii then set Action[i,$] to accept.

  (c) If [A →u•av, b] is in Ii and succ(Ii, a) = Ij, then set Action[i,a] to shift j (a must be a terminal).

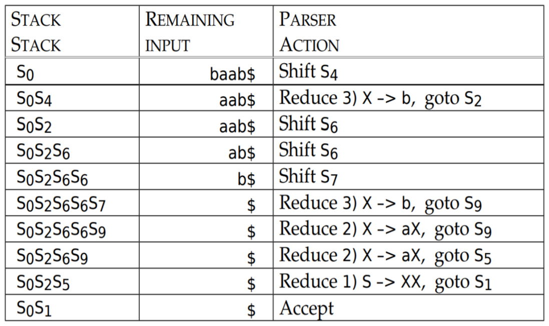
parse tables.png



State on top of stack	Action			Goto	
	a	b	$	S	X
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Figure 2: LR(1) parse tables for the grammar given in section 6.2 [7].

parse.png



Stack Stack	Remaining input	Parser Action
S_0	baab$	Shift S_4
S_0S_4	aab$	Reduce 3) X -> b, goto S_2
S_0S_2	aab$	Shift S_6
$S_0S_2S_6$	ab$	Shift S_6
$S_0S_2S_6S_6$	b$	Shift S_7
$S_0S_2S_6S_6S_7$	$	Reduce 3) X -> b, goto S_9
$S_0S_2S_6S_6S_9$	$	Reduce 2) X -> aX, goto S_9
$S_0S_2S_6S_9$	$	Reduce 2) X -> aX, goto S_5
$S_0S_2S_5$	$	Reduce 1) S -> XX, goto S_1
S_0S_1	$	Accept

Figure 3: LR(1) parse process on the string of the grammar given in section 6.2[7].

(3) The goto transitions for state i are constructed for all non-terminals A using the rule: If succ(Ii, A) = Ij, then Goto [i, A]= j.

(4) All entries not defined by rules 2 and 3 are errors.

For example, let us consider the following simple grammar:
0) S' ⟶S
1) S ⟶XX
2) X ⟶aX
3) X ⟶b
Figure 2 illustrates the parsing tables that might be generated for this grammar under LR(1) methods, and Figure 2 traces the parsing of the simple string "baab" in this language (The $ symbol is merely a null signifying the end of the input stream).

6.3 States, Stacks, and Shifts

We have now at this stage briefly outlined the fundamental components of LR parsers with 1 unit of look-ahead. We have also built parsing tables to control the operation of these parsers. Unfortunately, between shifting states onto the stack and reducing states, we may encounter grammatical situations where we may need some delicate footwork to conduct a successful parse. To do so, we must first define some additional terms.
For a given non-terminal B, the first set FIRST(B) is the set of all terminals that B may begin with. The follow set of B FOLLOW(B) enumerates the set of all terminals that B may be followed by. These sets are used to resolve shift-reduce and reduce-reduce conflicts in the input stream.
A shift-reduce conflict occurs when it may be unclear at a certain point in the parsing process whether to shift or reduce, as in the following case:
P ⟶ *id*•
P ⟶ *id*• (id)
The use of a single-token look-ahead allows us to look one token ahead of our current state, and compare what we find to the follow set of the element we have just pushed onto the stack. In the case above, if the next terminal found by the look-ahead is "(" , then we know to continue shifting. If otherwise, we reduce.
A reduce-reduce conflict, on the other hand, occurs when it may appear to be possible to reduce a single state in multiple ways. In the following example, for instance, we might be unsure as to which non-terminal to reduce to:
P ⟶ *id*•
K ⟶ *id*•
Supposing part of our grammatical definition is
$S \longrightarrow P|K + B$,
we can once again compute the follow sets of P and K and compare to the result of peeking 1 token ahead, which tells us that if the next terminal is "+", we reduce to K, otherwise reduce to P.
However, using look-ahead alone is not enough. Suppose we have the following productions appear in our grammar:
$P \longrightarrow S\bullet$
$K \longrightarrow S\bullet = L$
$S \longrightarrow id\bullet$
$S \longrightarrow *L$
$K \longrightarrow *L = L$
$L \longrightarrow P$
In this case, supposing the top state on our stack is S, we have a shift-reduce problem. Depending on whether we got S by reducing "*L" or "id", we need to either reduce or shift. Simply looking ahead and at the follow sets of S doesn't help, because if we reduce to P and S was generated from "id" while the look-ahead is "=", then reducing to P would result in the sentential form K -> L = L, which does not exist in the grammar. Therefore we must shift, and we can only break this conflict in such a way if we have maintained some left context about the symbols we have seen, in addition to maintaining a look-ahead. How this is done is a matter of implementing a simple look-up, as unless a conflict is reached this left context will not need to be used.

7 LALR PARSING PROCESS

LR(1) parsers are quite powerful, but as their parse tables are built from unique configurating sets of the given grammar, they tend to have exorbitantly large parse tables. LALR parsing promises to alleviate this problem by merging similar states in the tables, while maintaining much of the power of LR(1) look-ahead systems. Essentially, LALR differs from LR(1) only in the size of the tables. As a somewhat optimized subset of LR(1), LALR(1) systems are the most commonly used in industrial parsers and parser generation, and will also be the parsing method we pursue in our own project.

7.1 LR Table Reduction

There are two primary methods of generating LALR(1) parsers. The first is brute force, where we generate the tables in exactly the same manner as in LR(1), and then compress by finding and merging similar states. Unfortunately, this method is quite expensive when generating the parser, more so than setting up a naive LR(1). Instead, we will use step-by-step merging, which searches for similar states with each new closure computed, and merges on the spot while the tables are being built. Since we are compressing on the fly as the parser is generated, the space requirements for step-by-step LALR are always better than those for LR(1).

7.2 YACC

One widely used parser Unix users will be familiar with is YACC, the Yet Another Compiler Compiler. YACC takes in the input stream and some form of lexical analysis and picks up the tokens given by said analysis in the stream. Grammar rules give structure to the tokens. When a valid rule for a set of tokens is recognized, then program-defined action is taken [3]. YACC scans code left to right, and in doing so can recognize malformed input before execution of anything else. YACC is often able to skip beyond the erroneous elements of the input and continues parsing the input stream save for serious violation of grammar [3].

The specification of rules and tokens is left up to the user, and as such nearly any language that follows the lexical analysis restrictions can be parsed. This is sort of similar to how a define type (grammar) and EBNF (lexical analysis) could be used in racket to construct a parser generator. Much of the inspiration for our final milestone has been drawn from YACC.

8 THE PARSE MASTER 9000 AKA YAPP

8.1 Project Description

With this deep understanding of how parsers walk through an input string, our team is moving forward to the next milestones for the construction of the Yet Another Parser Parser. The purpose of the parser generator is to move up a level of abstraction, so to speak. It is better to write code that can handle more general input, and the cost and time savings of not writing a parser for every single different user defined grammar is immense. Racket provides some very unique tools that can be used in our generator. Define type and the user defined EBNF will give the parser the grammar (production rules) and the tokens of the language. From there, all that is left is construction of action and go-to tables and executing the construction of the parse tree. The advantages of LALR have been outlined, as have the examples that we will draw inspiration from.

8.2 Value and Impact

Proper parsing is instrumental in execution of code. Assuming a perfect implementation of a language, all it takes to incorrectly execute some input is to parse it incorrectly. If a malformed list makes it through parsing and the (rest list) is accessed in the implementation, then the input stream made it through the compilation process but is not performing according to the expected grammar. As a programmer, this could make debugging incredibly difficult, since improper inputs are allowed to leak through and could be executed. If this were occurring over a variety of odd inputs in a large language, there could be hugely unexpected results. The power of parsing is removing this serious danger. Only valid inputs will survive parsing.

In generating our own parser, we are partway through constructing our own compiler for abstract inputs. The value of a parser as abstract as our final milestone is that it can accept such a massive variety of user defined languages that the compiler could work for an equally large variety. In generating a parser, hand writing individual cases

9 SUMMARY AND FINAL THOUGHTS

Parsing at its core is a very simple task: recognize valid inputs and feed them to the rest of the compiler. However, the underlying difficulty lies in how abstract a parser can be made. While it is relatively simple to write a case for each individual element of one specific language with limited grammar, it is very difficult to construct one that can take nearly any grammar and tokens and create a parser robust enough to handle all the valid inputs of the given language specification.

Armed with knowledge of the different parsing approaches and the usefulness of parser generators at compile time, our team is now able to move forward with development of our parser generator.

10 ACKNOWLEDGEMENTS

Our team would like to express our thanks to the entire CPSC 311 course staff, specifically our instructor Dr. Wolfman, and project facilitator Ziyang.

REFERENCES

[1] Alfred V. Aho. 2014. *Compilers: principles, techniques, and tools.* Pearson.
[2] Ceriel J.H. Jacob. 1985. Some Topics in Parser Generation. *Vrije Universiteit* (1985). https://www.cs.vu.nl/~ceriel/LLgen.html
[3] Stephen C. Johnson. 1978. Yacc: Yet Another Compiler-Compiler. http://www.cs.man.ac.uk/~pjj/cs212/yacc/yacc.html
[4] Mohtashim. 2014. Compiler Design Tutorial. https://www.tutorialspoint.com/compiler_design/index.htm
[5] Nico. 2018. Lexical Analysis - (Token|Lexical unit|Lexeme|Symbol|Word). https://gerardnico.com/code/grammar/token?do=edit
[6] Rangra Rachana and Asst. Professor Madhusudan. 2015. BASIC PARSING TECHNIQUES IN NATURAL LANGUAGE PROCESSING. *chana Rangra et. al., International Journal of Advances in Computer Science and Technology* 4, 3 (Mar 2015), 18–22. https://pdfs.semanticscholar.org/12b5/f0f60c09187daf51c08c414ffcc7aea31707.pdf
[7] Keith Schwarz. 2012. CS143 Complilers. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/?fbclid=IwAR3wIbymVG6ZpyYt_UInB13hhRK3IEdL_4A8rpQnD9AGWmbed2_-8flHveo
[8] Elizabeth Scott and Adrian Johnstone. 2018. GLL syntax analysers for EBNF grammars. *Science of Computer Programming* 166 (2018), 120âĂŞ145. https://doi.org/10.1016/j.scico.2018.06.001

[9] Wikipedia contributors. 2018. Left recursion — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Left_recursion&oldid=862411387 [Online; accessed 8-November-2018].