Plan/Proof-of-Concept: 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 FROM PROPOSAL

This proof, combined with the attached code form the 90 percent milestone for this project. We aim to construct a fully functioning parser generator, similar to YACC or The Chicken Parser using Racket. User defined input will be combined with dynamic code that constructs and compresses tables, before constructing and returning a stack based parser implementing the generated tables and given grammar.

The skeleton for our code generator includes three major areas, and a few potential areas for extension. User input must be defined, constrained, and used appropriately. Extending this, the data type the user gives could be automatically generated from an EBNF grammar. This information is then used to construct the required tables, which are returned to the final code generation function.

This function is the completed parser. Using the tables, the generated code is able to take an input string from the given language and turn it into an abstract representation for the rest of the compilation pipeline.
These are the first complex elements of the whole parsing process. The difficulty in this project is not designing code generation for one language, but attempting to do it for any sufficiently simple language. The level of abstraction needed to be able to handle different types of recursion, symbols, operator order, and the like will be extreme. Completion of all elements of the program will constitute the simplest suitable approach, while the completionist approach will include various implementation modifications designed to handle more complicated inputs, on the fly compression, and runtime optimization.

2 BACKGROUND

LALR parsing has many complex elements that must be carefully considered when it comes to implementing the final program. While the typical match-based parsing in Dr.Racket would be possible for any of the languages we are examining, these solutions are not abstract enough, nor is LALR parsing implemented. The overall implementation of this parser is complicated and potentially costly, mostly due to the construction of configurating sets. Additionally, a custom stack implementation needs to be developed in order to handle the states and goto tables. This stack will use the aciton and goto tables in order to determine what next step to take, be it pushing another state onto the stack, or popping off the current state and moving forward.

2.1 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 [12].

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 non-terminal 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.

2.2 Action/Go-to Tables

Taken from our original report, but absolutely critical to the implementation warranting inclusion in entirety. 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 [12].

(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).

(3) The goto transitions for state i are constructed for all nonterminals 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

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 1: LR(1) parse tables for the grammar given in section 6.2 [12].

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 2: LR(1) parse process on the string of the grammar given in section 6.2[12].

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).

3 IMPLEMENTATION

(Note: Here's the Github Repository link to our project: https://github.com/lemonhos/parse-master)

There are three major areas that require attention: the initial user construct, table construction, and final stack-based parsing.

Initially, users must hand this parser a specific set of grammar that defines what is allowed in the language. From this set of grammar and production rules, the action and goto tables can be generated. These tables will be the instruction set for export to the final

parser. The actual parse function takes these tables and uses it to generate the final stack-based parser. The output of the program will be a parser that utilizes the constructed table and our stack implementation in order to take in a token string and output the appropriate abstract syntax for the user-given language.

3.1 EBNF and User Grammar Creation

Although we aim to make the parser valid for all valid EBNFs, the user must still input a valid grammatical structure including terminals and production rules. Borrowing heavily from [3]. This structure is the main input for the parser and is structured this way to give access to the rules for construction of the needed tables. Terminals are critical for preventing recursion in the next step, and can be accessed easily as well.

```
(define given_Grammar
    (parse-helper
        (terminals (list <given terminals>))
        (production_rules
            (list of production rules))))

;; production rules take the form of
    (production_rule_name (rule))

    example: (plus (AE + AE))
```

Both terminals and production rules will be separate defined types in Racket. From the given grammar, the parse-helper receives the information needed to move forward in the parsing process.

3.2 Building Action Tables

The set of production rules given to the parser are used in the main parsing program to construct the action tables. The action tables can be constructed recursively from the given rules

```
(define (build-action-table production_Rules)
    (calculate_config_sets production_Rules)
```

3.3 Configurating Sets, Successors, and Table Elements

The process of building the action table involves first determining all possible states that might be found on the stack and all possible inputs that the parse stack might encounter. We must do this by computing a set F = {I1, I2, I3,....,In} of all our states, where every state Ii corresponds to a unique configurating set. To this end, we find every possible combination of productions and positions that might describe how far we have gotten into parsing a particular production. For every combination of production and position we can find a configurating set using the following function.

```
(define (build-closure production position)
    ...)
```

An example of this is given in section 2.1. In this example the production in question is $A \rightarrow X \bullet YZ$ and the position value is 1, implying the next token to be parsed is the second token of the production, as indicated visually by the black dot. Given this production and this position, build-closure will then compute the configurating set given in the example. The broad conceptual elements of computing each closure are described at the end of section 2.1.
Computing F and accumulating a set G of all possible input tokens allows us to sketch the rows and columns of our action table as in Figure 1, but we must now populate the cells as necessary.

To accomplish this, we construct an implementation adhering to part (2) of the table algorithm described in section 2.2. In short, we may iterate over every [state, nextInput] of the action table we have made and fill in the position with one of shift(k), reduce(k), or accept(), where k refers to the state we will push onto the top of the stack once the prescribed action has been executed. If none of the conditions illustrated in part (2) of the aforementioned table algorithm are satisfied for a given cell, we will fill that cell with error(). Should the parser ever arrive at that particular location in the action table during a runtime parse, the input string is grammatically invalid and the parser will halt.

There are a few noteworthy implementation details to be considered when building the action table. Every configurating set, or state, must be referred to in the action table, and as such we maintain a global map F of configurating sets, each with unique identifiers as keys. Configurating sets are added to this data structure as they are discovered in build-closure(), allowing for fast state lookup throughout the table-building process. In addition, as build-action-table() iteratively calls build-closure(), for each production the position is initialized at 0 and incremented with successive calls to build-closure() until all configurating sets of the production have been found. This allows us to maintain a 1-state lookback memory within build-action-table(), such that if state I4 is computed and then state I5 is computed, and I5 is the successor of I4, the lookback memory will contain the identifier key of I4. This allows us to create another global map S, containing key-value pairs of states and their successors. In the given example, searching the identifier for I4 in S would return its successor, I5. With this strategy, once we have built F and S, the entire Action Table can be filled in merely by looking up keys in these datasets.

3.4 GoTo Tables

With the elements required for building Action Tables in place, constructing a GoTo table for a given grammar is somewhat trivial. Similar to the action tables, GoTo tables are drawn along one dimension with the identifiers of all possible states. Unlike action tables, where columns are based on possible inputs to the stack, the columns of the GoTo denote the non-terminals of the grammar. As an example, refer to Figure 1. As the global sets S and F are already constructed at this point, drawing the GoTo table consists merely of iterating over the nonterminals and states of the grammar. This is implemented as the first step in the following GoTo-building function.

```
(define (build-goto-table ...)
    ...)
```

Filling in the GoTo table is then a matter of iterating over the cells in the table and looking up the successors of states in S. A description of the algorithm used is given in part (3) of the table algorithm given in section 2.2. As before, any cells not filled in with

states throw an error if they are ever reached by the parser during runtime.

3.5 Stack + Table = Parser?

After we finish creating our stack of input tokens and constructing action and goto tables, we need to export those generated results into external files for reuse in the future. One of the exported files should be the parser itself, and it contains stack of input tokens and lexical analyzer, and allows input/output. The action and goto tables should also be exported separately, as they are considered inputs to the parser and differ by specifications.

4 LOW-RISK APPROACH PLAN

The low risk approach will include a fully functional version of the parser generator, without the level of abstraction expected in the high risk approach. From a user given EBNF and grammar, the production rules will be derived and used to build the action tables, but in this approach, we will restrict the grammar to be a small since it will make the implementation easier, and we will try implementing the our project to support general grammar in the high-risk approach. After action and goto tables are generated, then the tables will be exported to a separate file.

Stack given list of tokens in input string. Push states onto the stack, top element corresponds to cell in the exported action table. Shift state onto the stack, and continue to check the state on the top of the stack.

Write stack based parser implementation, use the generated tables as lookup and gotos for the states that are being popped off the stack.

5 HIGH-RISK APPROACH PLAN

The high risk approach will include all features included in the implementation section. Aside of those, we are going to implement the parser to support the unconstrained version of EBNF with a more effective compression function as the key of LALR.

5.1 General EBNF

Since we decided to implement our project to support restricted grammar in low-risk approach in order to lower the difficulty, we need to implement our project to support a more general EBNF grammar in this approach. By doing so, our project can be applicable to general programming use. To do that, we will add extra rules in addition to those we write in the low-risk approach in our helper functions to handle the additional grammar rules that users provide to our program.

5.2 Handling Ambiguity and Left Recursion

In low-risk approach, we didn't mention how to handle ambiguities and left recursions which are special forms of parsing tree. However, since these forms aren't generally accepted by many languages, we need to eliminate these two forms or reduce them to applicable states.

Ambiguities fail to convert the parser tree into an LR tree. By definition, ambiguity in compiler design means a grammatical input can generate two parse trees. Ambiguity happens because of the following reasons and we have corresponding solutions for them:

(1) Precedence: The operators are not ranked. To solve ambiguity, we need to set up the precedence list to force the certain trees to be unambiguous.
(2) Associative: sequence of identical operators can group either from the left or from the right. In a grammar with many associative operators, it can be ambiguous, and it can be fix by only insist on grouping in left.

When implementing high-risk approach, we will add an additional method to check and handle these two situations.

Left recursion is mentioned in background report. Here is an example used in background report, 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 run in an infinite loop, but it can be fixed in multiple ways including rewriting the offending production to A $\longrightarrow$Aa| B. A will finally be replaced be B after producing a sequence of a's (zero or more) [1]. By the time we implement the parser generator, we will use if statement to avoid the left recursion and fix it by the method above.

REFERENCES

[1] Alfred V. Aho. 2014. *Compilers: principles, techniques, and tools.* Pearson.
[2] A. V. Aho and S. C. Johnson. 1974. LR Parsing. *ACM Computing Surveys (CSUR)* 6, 2 (Jun 1974), 99âĂŞ124. https://dl.acm.org/citation.cfm?id=356629
[3] Dominique Boucher. [n. d.]. Chicken Scheme. http://wiki.call-cc.org/eggref/4/lalr#the-grammar-format
[4] Stephen Jackson. 2009. A Tutorial Explaining LALR(1) Parsing. https://web.cs.dal.ca/~sjackson/lalr1.html
[5] Ceriel J.H. Jacob. 1985. Some Topics in Parser Generation. *Vrije Universiteit* (1985). https://www.cs.vu.nl/~ceriel/LLgen.html
[6] Stephen C. Johnson. 1978. Yacc: Yet Another Compiler-Compiler. http://www.cs.man.ac.uk/~pjj/cs212/yacc/yacc.html
[7] Aleksey Kladov. 2018. Modern Parser Generator. https://matklad.github.io/2018/06/06/modern-parser-generator.html
[8] Mohtashim. 2014. Compiler Design Tutorial. https://www.tutorialspoint.com/compiler_design/index.htm
[9] Nico. 2018. Lexical Analysis - (Token|Lexical unit|Lexeme|Symbol|Word). https://gerardnico.com/code/grammar/token?do=edit
[10] Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. [n. d.]. Lexer and Parser Generators in Scheme. https://www.cs.utah.edu/plt/publications/scheme04-ofsm.pdf?fbclid=IwAR2lGldElKC9BYf2nzONPVUfHZPstUAER58wABWlE9icNQ2ppbTc5Oz1lno
[11] 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
[12] Keith Schwarz. 2012. CS143 Compilers. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/?fbclid=IwAR3wIbymVG6ZpyYt_UInB13hhRK3IEdL_4A8rpQnD9AGWmbed2_-8flHveo
[13] 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
[14] Wikipedia contributors. 2018. Formal grammar — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Formal_grammar&oldid=864957481 [Online; accessed 8-November-2018].
[15] 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].
[16] Jason Young. 2015. Code Generation: An Introduction to Typed EBNF. *All Graduate Plan B and other Reports* (May 2015). https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=1505&context=gradreports&fbclid=IwAR0VDxPPB4-qE3M8RXJJjS1hdImyQluCP4xTWWzeRQMwyaHKV_Yax0bvwCk

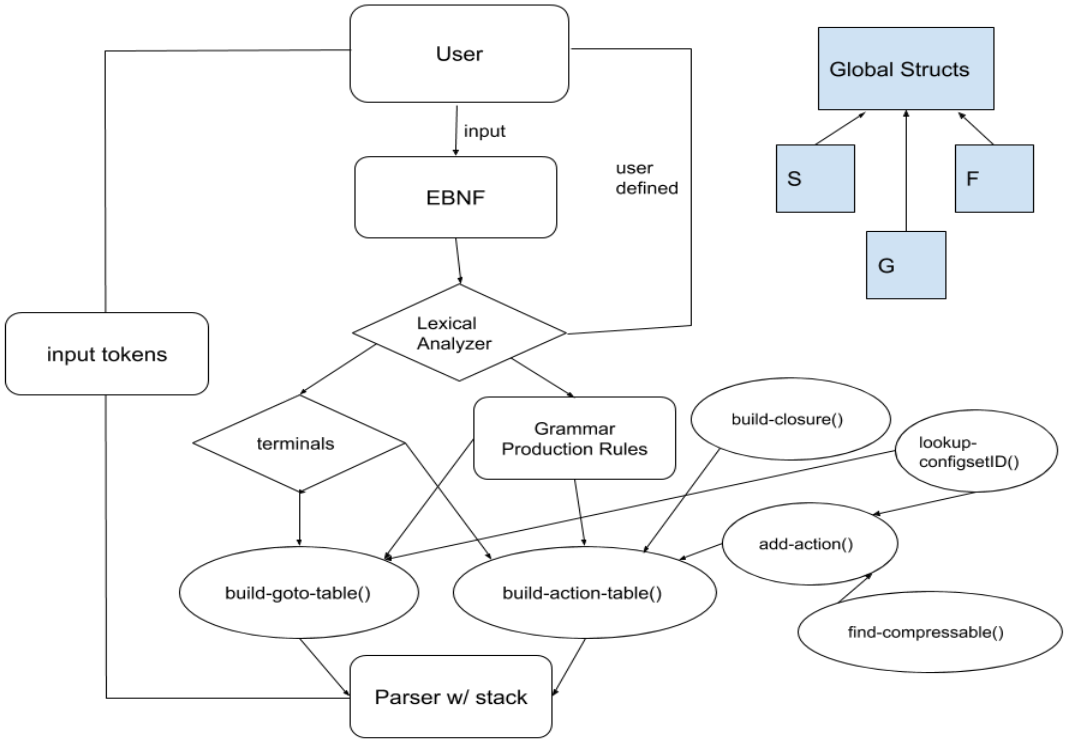

Figure 3: Key components of the Parse Master 9000 system