Proposal: 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 INTRODUCTION

As with every element of computer science, compile time is made up of a large sum of individual parts, each of which must be robust and correct. One example of these elements is parsing: the conversion of a given language's grammar into a compiler-usable output. While relatively easy to construct parsers for individual cases, making one abstract and robust enough for a variety of input grammars, such as any language defined by EBNF, is significantly more challenging.

Our chosen project type is number 5, with the specific domain we are operating in being that of parsing EBNF. With a plethora of other parser generators available in other languages typically C++, C, Python, and Java, we will construct our proof of concept in Dr.Racket. The usefulness of match syntax, along with the ability to generate define-type gives us some very interesting resources for both our final milestone and initial investigation into LALR parsers.

We aim to delve into the theory of parser construction and how the front end of compile time works, before constructing a small proof of concept. Parsers can take a wide variety of approaches to handling different grammars, and how syntax trees are created and traversed. Through examination of these approaches, we will design and implement a proof of concept. The aim of this proof of concept will be to construct an LALR parser generator for a given language's EBNF definition.

2 APPROACH

2.1 Plan to get to 80% Milestone

For the 80 percent project milestone, we will begin by introducing the theoretical context by which we will motivate our project development. A background summary of context-free grammars, with a high-level overview of their role in defining programming languages, will be provided alongside some of the strengths, pitfalls, and various notes of interest surrounding this class of formal grammars. With some of the theoretical support given, we can proceed to a summary of compiler construction, specifically with regards to the crucial roles that the parsing, lexical analysis, and semantic analysis steps play in the transformation of source text into viable machine level programs. Special attention will be given to the significant value of parser generators in streamlining the compiler design and implementation processes, thus motivating the remainder of the project.

We will extensively research the parsing of grammars into intermediate forms appropriate for the program generation stages of the compile process. We will cover important topics including formal grammar definitions, lexical analysis, parse trees, left recursion, tokenizing, symbol table generation, as well as high level parse approaches such as top-down LL(*), bottom up LR, or look-ahead LALR. Our analysis will be extended to the development of parser generators, specifically those that generate parsers from EBNF definitions of grammars. Some attention will be paid to the construction principles of some of the more common existing parser generators, including Yacc, Bison, and ANTLR.

To round out the background research report, we will explore some of the implications of developing a parser generator under the Scheme/Racket paradigm as it relates to our own project goals.

2.2 Plan to get to 90% Milestone

By the time we have reached this milestone, we will have conducted extensive study into the construction of LR and LALR parsers, and we will accordingly present an updated and completed outline of our design for our system. This will include a discussion of the parsing strategies we will be supporting, the design of our lexer/tokenizer, our design strategies for reading grammar specifications and constructing goto/action tables, and any hard constraints we will place

on the the set of acceptable grammars for our system. We will discuss our decision in adopting the LALR parsing methodology, as well as any implications or notes of value regarding the implementation of a parser generator in a Scheme/Racket paradigm.

In addition, we will present a skeleton implementation for our parser generator. While this proof-of-concept system will not be a fully operational parser generator, we expect it to be a partially functioning example of the general implementation approach we plan to follow to reach our 100% milestone. In particular, we expect to have the prototype implementations for the following:
- A lexer generator
- Generation of action/goto tables
- Lookahead mechanisms
- Built-in error handling for a limited set of lexical and semantic errors
- Interpretation of user-given EBNF grammar specifications
- Generation of simple parse trees

Furthermore, the system should ideally be sufficiently operable to use these components to generate a parser that parses some simple grammars into their correct abstract syntax trees. Anything more sophisticated, including handling of ambiguous grammars and user-directed define-types, will not be supported at this stage.

2.3 Plan to get to 100% Milestone

To reach the goal for our final milestone, we will finish the code implementation for our parser generator, which will be fully operational by this stage. In addition to generating action tables, interpreting EBNF grammars using LALR methodology and handling lexical and semantic errors according to our 90% milestone, the output parser generated from our program will also be able to handle complex cases such as ambiguous arguments and user-directed define-types, and set constraints to recursion rules. We will also discuss some justifications in implementing these functions for our parser generator, as well as their significance.

To demonstrate the output parser's functionality, we plan to illustrate how the syntax tree based on given EBNF input is generated under provided grammar rules. For input examples, we plan to use instances previously shown in 311 course. As a part of the final steps, we will show what the output will look like and how its format will be suited for compiler to consume in upcoming stages.

3 POSTER

For the poster, we plan to lay out some background knowledge to reveal the value of writing the parser generator, and we will try to show our progress by using flowchart. In addition to those descriptions, we are also going to include some visualizations of our LALR parsing approach, in the form of a bottom-up tree, to provide more insights on the mechanism's behavior and effectiveness. Also, we will include some code examples from lecture or homework as inputs to our parser generator and show the result parser on the poster as simple demonstrations of our 100% milestone. Finally, we will include some constraints of our generator, as well as the justifications for these constraints in our poster.

4 STARTING DOCUMENTS

Here are some resources we will use as starting points in our project:
1. Compilers: principles, techniques, and tools. [1]
2. Lexer and Parser Generators in Scheme [5]
3. Modern Parser Generator [4]
4. A Tutorial Explaining LALR(1) Parsing [3]
5. A course blog by Stanford University [6]
6. Two EBNF type related resources [7] [8]
7. Dr. Racket Language Standard [2]

5 SUMMARY

We are designing and implementing a parser generator for EBNF definitions in Racket/Scheme language using Look-Ahead LR mechanism, and demonstrate its various functions such as generating tokens and action tables, handling errors, interpreting user-defined types, building syntax trees based on EBNF specifications and setting constraints on recursions.

REFERENCES

[1] Alfred V. Aho. 2014. *Compilers: principles, techniques, and tools.* Pearson.
[2] Clarie Alvis, Yavuz Anderson, Leif Arkun, and et.al. [n. d.]. Racket Documentation. https://docs.racket-lang.org/
[3] Stephen Jackson. 2009. A Tutorial Explaining LALR(1) Parsing. https://web.cs.dal.ca/~sjackson/lalr1.html
[4] Aleksey Kladov. 2018. Modern Parser Generator. https://matklad.github.io/2018/06/06/modern-parser-generator.html
[5] 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
[6] Keith Schwarz. 2012. CS143 Complilers. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/?fbclid=IwAR3wIbymVG6ZpyYt_UInB13hhRK3IEdL_4A8rpQnD9AGWmbed2_-8flHveo
[7] 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
[8] 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