Rochester Institute of Technology

RIT Scholar Works

Theses

2006

Object-oriented LR(1) parser generation

Christopher Luckett

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

Luckett, Christopher, "Object-oriented LR(1) parser generation" (2006). Thesis. Rochester Institute of Technology. Accessed from

This Master's Project is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Project Proposal

Object Oriented LR Parser Generation

Christopher Luckett

Chairperson: Prof. Axel Schreiner

Reader: Prof. Rajendra Raj

March 23, 2006

Contents

1 Introduction **3**

2 Summary **4**

3 Functional Specification **6**

4 Architectural overview **8**

5 Deliverables **9**

6 Proposed Schedule **9**

7 References **10**

1 Introduction

LR parsing, known also as bottom up parsing, is the syntax directed analysis of input tokens from the left to right producing syntax trees of rightmost derivation. It is one the most general methods of parsing and allows for left recursion and allows for error detection which the top-down LL type parsers cannot. The LR parsing algorithm was first proposed by Knuth in 1965, in his paper, *On the translation of languages from left to right*. In that paper he describes the LR parsing algorithm and the automatic generation of parsers from a grammar.

Traditionally LR parsers are composed of four parts, a stream of input tokens generated by a lexical analyzer, a stack containing processed input and previous states of the parser, a table used for making decisions about input symbols and a driving algorithm which coordinates and responds to the respective states of the other three components. Overly simplified, the input token and the current state are used as indices for the table. The location specified in the table contains one of four actions: shift, reduce, accept, or error, and the next state for the parser. The parser then performs the task described by the action, then transitions to the state specified by the table.

Unfortunately, both the workings of the main algorithm and the generation of the state table tend to be less intuitive compared with recursive descent mechanism of the LL type parsers. Also, the presentations of this algorithm are generally difficult to understand by newcomers to the area of language processors, and the implementations, generated by tools, do not make concessions in favor of readability. This lack of readability, makes it difficult to see two area which tend to be difficult to understand: first being the how the lookahead sets which are generated by the parser-generator are used in calculating the parsing table and secondly how that table generated by compacting a much larger table which contains many redundant and unnecessary entries.

2 Summary

The overall objective of this project is to provide an alternative implementation of the LR(1) parsing algorithm, by replacing the the state transition tables with objects called markers, and the driving algorithm with execution paths distributed across these markers. At this point it should be noted that while I am eliminating the state transition table and extending the table interpreter across the markers, all possible markers will be generated by the parser-generator at the time that the parser is generated. Furthermore, the parser will contain all of these markers and the relationships between these markers will be static and predetermined. This setup I believe creates an analogue to the state transition tables rather than a congruency. Also this will provide a window to the table compaction portion of generation phase, i.e. that redundant markers would represent redundant states in the state table that could be removed. Three main goals will be pursued by this project, firstly to create an object-oriented representation of the LR(1) parsing algorithm, secondly to implement this algorithm, and finally to build a parser-generator which builds parsers of this type that are compatable with existing lexical analyzers and constructed from grammars in Backus-Naur Form(BNF) with similar syntax to that of currently existing parser-generators, for example YACC.

The first goal which is currently underway, and has been successful for grammars of the LR(O) type. Essentially from this point, the algorithm needs to be adapted to make use of available lookahead sets, to accomodate a wider spectrum of grammar, which are ambiguous in LR(0). Conceptually the algorithm involves making use of marker objects which are based upon the rules found in the grammar. These marker objects are chained together and are representations of the states of the parser. The input is passed to these chains, where the actual execution of the main algorithm occurs, parsing the input until it is accepted or rejected as a program in the language described by the grammar.

The second goal will be accomplished by the gleaning of information from the grammar. A comprehensive trace of parser states, justification of decisions made in certain places, current rules being examined, the meaning of specific states and some animation of the parsing process may be made available. This is possible in this scheme of parser, where it is not in traditional parsers, because the grammar rules are used in the analysis and not transformed into incomprehensible tables.

The third goal, that of the parser-generator, is the most important part of this project since it will allieviate the need for hand generation of the parsers which tends to be an exercise in tedium. The parser-generator will take as input a grammar in BNF, code to be added into the parser that is generated, and some directives which will determine what and how output is generated.

A more concise summarization of my plans is that I will modify the LR(0) algorithm, that I have already tested, into a LR(1) algorithm by incorporating the lookhead and follow sets. Then I will hand construct a LR(1) parser, using the object oriented approach, that accepts LR(1) grammars as the language. Finally, I will construct the backend into a translator that generates parsers from the accepted grammar.

3 Functional Specification

The parser-generator will accept as input a grammar defined in a language similar to BNF with some notable exceptions: first, as a preamble to the grammar it will be necessary to define the lexical specification and the lexical analyzer that is to be used. Some directives will be included to allow for both declarations.

The second exception is that following the preamble will be a section where the programmer can define public fields for the parser which can be manipulated within the grammar. These fields can be of any type and can be used for a wide variety of purposes. Such purposes may include fields for generating specific types of output, or fields for retrieving statistics about the parser.

The third of these exceptions is that in addition to defining grammar rules, it will also be possible to define actions that take place upon reduction, these actions will be defined in JAVA code with variables defined by the rules themselves, this allows for simple translators to be developed to test a parser for example. Note in figure 1 the area enclosed by curly braces, the meaning of that statement is the top of the stack ($$) is assigned with the sum of the values at the first position and the third position.

```
S:S+T
{$$=$1+$3}
|T
T:number
```

Figure 1: Sample Input Grammar

The final exception to a standard BNF grammar is that another section will be defined following the basic grammar which will allow a place for the programmer to enter JAVA code. This section is necessary to allow the construction of new classes that may be desired for use within the grammar. At this point, it is necessary to consider other tools which have been created for

similar purposes. One tool which has particularly similiar input constraints is *pj* created by Axel Schreiner, which many of the ideas for input in this project have come from. *pj* is a preprocessor that connects a parser-generator and a lexical analyzer and provides other useful features. So two options for the interface are available: first, set up the input to the parser generator so that it exactly matches the output of *pj*. The second option is just incorporate the ideas, especially the three part architecture, of *pj* into a new interface. These options will be weighed carefully throughout the implementation of the rest of the parser-generator, and perhaps both will be available.

The output of the parser-generator is the parser itself and is composed of two parts, the grammar representation, and the parsing mechanism. The grammar representation will be an objectified view of the grammar and the actions defined in the input a.k.a. the markers and the relationships between them. The parsing mechanism is the set pre-defined objects required for parsing and the programmatic interface.

The interface for the parser is simple, the parser being an object, is constructed with no arguements. The parser will contain methods that signal to parse with varying options pertaining to input and output conditions. Many variations will exist determining how much output and debugging information will be generated for any run of the parser.

The programmer is expected to use this parser within the the context of another application, so if a stand-alone parser is required a wrapper program should be implemented.

4 Architectural overview

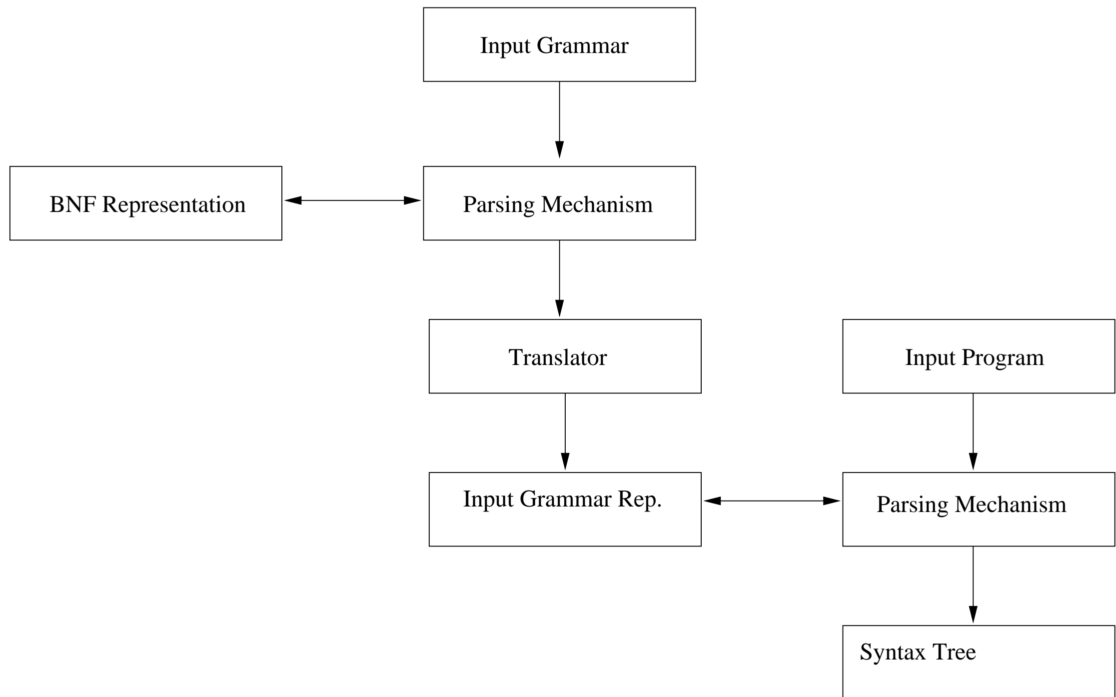

Figure 2: Overall Architecture of Parser-Generator

Since the parser-generator is just a parser with a back-end that generates parsers, the architecture is cyclical in nature. Essentially the parser-generator will contain an object that represents a grammar for BNF-style grammars which it will use to parse other BNF-style grammars that are the input grammars for parsers to be generated.

Simply put, the parser-generator will replicate its own parsing mechanism except that the internal representation of the grammar will be of the input grammar instead of the grammar for BNF, and if a back-end is specified by the input grammar then it will be generated as well.

The architecture of the parser itself is not as cyclical as the generator, it is composed of a representation of the grammar it accepts and a factory for

generating markers and chaining the markers together providing state for the parser. The general process that the parser undergoes is that a base marker is created representing the starting rule of the grammar, for each non terminal marker new markers are generated expanding that rule. Then an input token is compared to each terminal marker, if there are any matches, those markers generate new markers which represent a further progression through the rule. When an end of rule marker is generated, and no other markers are generated, the rule reduces the input and converts it to nodes on a syntax tree. This process continues until either all input is accepted or the input runs out in error.

5 Deliverables

This project will include, a report describing the parser-generator and how it was developed, a User Manual, describing how to use the parser-generator, some examples showing both how to use the parser-generator and that it is capable of generating parsers and finally the code for the parser generator.

6 Proposed Schedule

1.Modifying Current algorithm: 2 weeks.

2.Constructing Parser Generator: 5 weeks.

3.Writing Supporting Documents: 3 weeks.

4. Preparation for defense: 1 week.

Estimated Defense 10-11 weeks from date of submission. Most likely, however, since my chairperson is absent during the summer quarter, if the project is not complete at the end of the spring, then the defense will happen in the fall.

7 References

Alfred V. Aho, Ravi Sethi, Jefferey D. Ullman. Compilers Principles, Techniques and Tools. Addison-Wesley 1988.

Aho, A. V. and Johnson, S. C. 1974. LR Parsing. ACM Comput. Surv. 6, 2 (Jun. 1974), 99-124.

Frank DeRemer , Thomas Pennello, Efficient Computation of LALR(1) Look-Ahead Sets, ACM Transactions on Programming Languages and Systems (TOPLAS), v.4 n.4, p.615-649, Oct. 1982

D. E. Knuth. On the translation of languages from left to right. Information and Control, 8:607-639, 1965

Peter Pepper. LR parsing = grammar transformation + LL parsing: Making LR parsing more understandable and more efficient. Technical Report 99-5, TU Berlin, Apr 1999.

Pijls, W. 2000. LR and LL parsing: some new points of view. SIGCSE Bull. 32, 4 (Dec. 2000), 24-27.

Schreiner, A. T. pj(language processing).1.5, Pittsford, December 2005. http://www.cs.rit.edu/ ats/projects summary.html . Accessed 3/23/06.