CS3230R
Project Report

Implementation of an SLR(1) Parser

By

Darius Foo
A0097282W

Department of Computer Science

School of Computing

National University of Singapore

AY 2013/14 Semester 2

Advisor: Assoc. Prof. Leong Hon Wai
Deliverables:

- Report
- Source Code
- Readme
- JAR binary

Abstract

In this report, the implementation details of a parser for a class of context-free grammars, LR(k), are studied and elaborated on. A working parser for a simple list-processing programming language is developed from this research.

Subject Descriptors:
Programming Languages

Keywords:
parser, algorithm, implementation

Implementation Software and Hardware:
JRE 1.7.0, Java HotSpot(TM) VM

Table of Contents

Chapter 1

Introduction

The problem of parsing grammars is an essential one in the field of programming languages. In this project, the implementation of a parsing algorithm for a particular class of context-free grammars, LR(k), is studied.

1.1 Background

Context-free grammars are of great practical interest to computer scientists as they can be easily and efficiently parsed, yet are expressive enough for most purposes. A specific type of context-free grammar, named LR(k), exhibits a number of useful properties that make it well-suited to the implementation of programming languages (Aho, Lam, Sethi, & Ullman, 2007).

The name LR is an acronym: the L stands for **left-to-right** (the direction in which such a parser would process the input stream), and the R stands for reversed **rightmost-derivation** (alluding to how rules of the grammar are applied by a parser to reduce the input into a syntax tree). The numeric qualifier, k, is the number of tokens of *lookahead* symbols that the parser is allowed to peek at to decide how to reduce the input, and is typically 0 or 1. As there are many variants on the LR algorithm, other qualifiers may precede the name; the specific variant referred to here is the Simple LR(1) parser.

1.2 Project Goals and Scope

The goal of this project is the implementation of a Simple LR(1) parser-generator: a program that accepts as input an LR(1) grammar and produces another program which is able to parse sentences derived from that grammar. An analysis of the internal workings of the LR parsing algorithm will be provided, followed by the discussion of various implementation issues encountered.

Chapter 2

Problem Statement

The problem of translating an LR(k) language into different forms for interpretation or compilation can be formulated as the problem of parsing an LR(k) grammar.

2.1 Properties of LR(k) Grammars

LR(k) grammars are a class of context-free grammar that admit two main properties (Knuth, 1965):

- "LR(k) grammars can be efficiently parsed with an execution time essentially proportional to the length of the string."
- "A language can be generated by an LR(k) grammar if and only if it is deterministic, if and only if it can be generated by an LR(1) grammar."

The first statement means that sentences derived from LR(k) may be parsed unambiguously and efficiently, in a single pass without backtracking. The second states that an LR(k) grammar may be reduced into an equivalent LR(1) grammar, meaning that LR(1) grammars are no less expressive than LR(k) grammars.

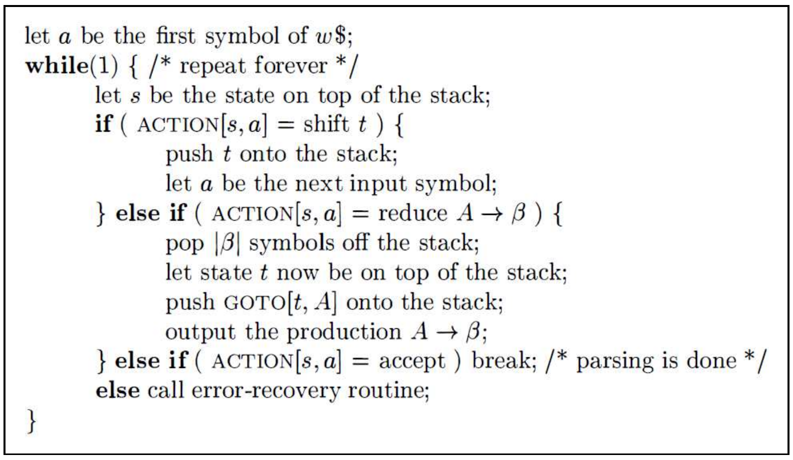
A parser takes as input a stream of tokens and produces an abstract syntax tree (AST) capturing the structure of the input program. The SLR(1) parser is no different, and thus the problem to be solved is to decide which grammar rules should be applied to reduce the input, given the tokens seen so far, and how these reductions should be carried out.

Chapter 3

Description of Algorithm

3.1 The LR Parser Algorithm

The LR parsing algorithm is as follows (Aho et al., 2007):



```
let $a$ be the first symbol of $w\$$;
while(1) { /* repeat forever */
        let $s$ be the state on top of the stack;
        if ( ACTION[$s, a$] = shift $t$ ) {
                push $t$ onto the stack;
                let $a$ be the next input symbol;
        } else if ( ACTION[$s, a$] = reduce $A \to \beta$ ) {
                pop $|\beta|$ symbols off the stack;
                let state $t$ now be on top of the stack;
                push GOTO[$t, A$] onto the stack;
                output the production $A \to \beta$;
        } else if ( ACTION[$s, a$] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

LR parsers are bottom-up *shift-reduce* parsers which operate via only two principal actions:

- A *shift* action takes a token from the input stream and pushes it onto the *parse stack* as a singleton AST node.
- A *reduce* action applies a grammar rule to the topmost AST node(s) on the *parse stack*, transforming them into other kinds of AST nodes.

The *parse stack* is the primary data structure of the parser. It contains nodes of the AST which the parser builds. When the input is completely consumed, only a single AST node will remain on the parse stack: the root node representing the entire program. Thus the AST is constructed in a bottom-up fashion.

The LR parsing algorithm is very simple at a glance. What complicates things is how the decisions to shift and reduce are made. This is also the primary distinction between variants of LR parsers.

A naive way to make these decisions would be to check every rule of the grammar each time a token is shifted for a match, and pick that rule to perform a reduction. If no rule is matched, perform a shift and repeat. Unfortunately this method would take more and time as the size of the input grows and will not scale well.

To avoid having to match every rule at every juncture, the SLR(1) parsing algorithm performs a copious amount of preprocessing, producing a static *parse table* which summarizes all information required to make a deterministic decision at every point. The GOTO and ACTION data structures in the algorithm above are columns of this table. Thus, assuming that table lookups may be done in O(1) time, the algorithm is guaranteed to terminate in time linear to the size of the input.

A visualisation of the parser follows (Aho et al., 2007):

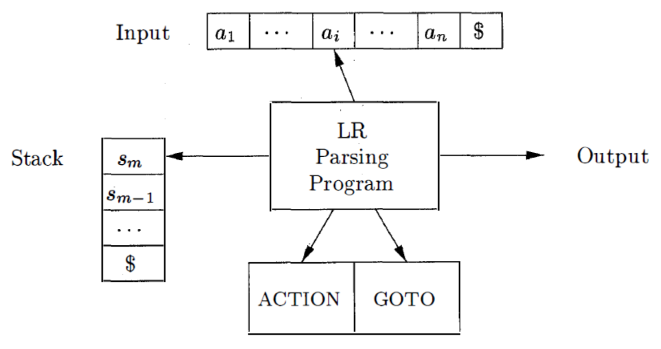


Figure 4.35: Model of an LR parser

The structure of the parser is seen clearly here: the parse stack is to the left, the input stream is on top, and the GOTO and ACTION columns of the parse tables appear at the bottom.

3.2 Producing the Parse Table

A key observation that leads to being able to summarize all information required in the parse table is that parsing is deterministic. Thus there is a finite number of states that the parser may be in with respect to the parse stack. These states may be derived from the rules of the grammar by inserting a special marker into every possible position on the right-hand side of the rule, shown here using an example grammar rule:

$$\text{Sums} \to \bullet\ \text{Sums} + \text{Products}$$

$$\text{Sums} \to \text{Sums}\ \bullet + \text{Products}$$

$$\text{Sums} \to \text{Sums} + \bullet\ \text{Products}$$

$$\text{Sums} \to \text{Sums} + \text{Products}\ \bullet$$

Everything to the left of the marker represents something the parser has pushed onto the parse stack. Everything to the right is still in the input stream. In this way each marked rule (called an *item*) summarises the state of the parser at any instant, though not completely - multiple rules for reduction may apply for a particular item. Thus the states of the parser must be characterised by *sets* of items.

In addition, these item sets should be *closed* to ensure that every relevant item is accounted for. The *closure* of an item set is the item set extended by any items produced by recursively expanding nonterminals following the marker.

3.3 Populating the State Transition Table

Each closed item set is now a complete summary of a distinct state of the parser. Once all closed item sets are enumerated, transitions between them (equivalently, between parser states) can be determined.

For each pair of item sets/states A and B, a transition from A to B should occur if for an item in A, the symbol following its marker appears on the left side of B.

These transitions may be stored in a table. They express exactly how the parser proceeds between states as it performs shift and reduce actions.

3.4 Populating the Parse Table

A few final reductions are performed on the state transition table (Aho et al., 2007):

- Columns for nonterminals form the GOTO columns of the parse table. They correspond to state transitions following reduce actions.
- Columns for terminals form the ACTION columns of the parse table. They correspond to shift actions.
- An extra column for an end-of-input symbol is added to ACTION columns; it indicates the parser having reached a terminating state.
- Item sets that contain items that are fully advanced (of the form $\mathrm{A} \rightarrow \mathrm{W} \bullet$) and are reducible by applying a rule have corresponding rows in the table populated with reduce actions using that rule. This only applies if the left side of the rule may follow A in any derivation.

The table now contains actions for every state in which the parser may legally be, for every token it might legally encounter in that state. If the table does not contain an action for any combination of state and token, a syntax error has occurred and the parser should report an error.

Chapter 4

Complexity Analysis

LR parsers are deterministic, and produce a single correct parse without guesswork or backtracking. This follows from the property that LR(k) grammars are unambiguous; given the required number of lookahead tokens, the correct derivation may always be made. Assuming that:

- Lookups in the parse table are done in constant time
- Construction of AST nodes and stack operations are done in constant time
- A token may be reduced as many times as there are rules in the grammar in the worst case

... the parsing algorithm takes O(mn) time, where n is the size of the input string, and m is the number of rules in the grammar. In practice, m is a relatively small constant, so the parse will effectively take O(n) time.

LR parsers involve a lengthy preprocessing step, however. To generate the parse table, assuming that:

- Closing item sets requires O(mi * m) time in the worst case, where i is the length of the longest right side of a grammar rule
- Populating both tables requires O(st) time, where t is the number of distinct symbols in the grammar and s is the number of parser states

. . . the preprocessing step takes O(2st + im^2) time, which is fairly reasonable as s, t, and i are all relatively small even for very complex languages.

Chapter 5

Experiment Setup, Results, and Observations

5.1 Toyscript

To test the parser, an interpreter for a small ancillary language called Toyscript was first written. Toyscript is functional enough for list processing, and complicated enough to test the robustness of a parser implementation: it contains prefix and infix operators with 8 levels of precedence and a good number of syntactic constructs.

A placeholder frontend was written for the interpreter while the SLR(1) parser implementation underwent development: CUP and JFlex, tools for parsing and lexical analysis respectively, were used. The interpreter was first determined to be stable via unit tests.

Subsequently the parser was implemented. High-level class-based representations for grammar rules and item sets were used, eschewing performance for intuitiveness and readability. The Toyscript grammar was encoded directly in the parser; while it can be easily changed and replaced, there is no way to modify it from outside the source code.

Once the SLR(1) parser was able to perform reductions on Toyscript syntax, it was integrated into the existing interpreter in place of CUP and tweaked to satisfy unit tests.

5.2 Results

The parser is currently able to handle all constructs in the Toyscript language. It passes all test cases provided with the code, which involves common functional programming examples:

- (Tail-)recursive functions
- Higher-order functions map, fold, filter
- Demonstrations of lexical scoping and variable shadowing

5.3 Observations

The SLR(1) algorithm was chosen because it provided the right compromise between expressiveness and simplicity of implementation. LR(0) ended up being too restrictive for some of the recursive constructs in Toyscript. LALR(1), a more expressive class of grammars, required a much more sophisticated implementation.

Due to limitations in the expressiveness of SLR(1), tweaks to the grammar had to be made to accomodate it. In particular, a distinction between an *expression* and a *value* had to be made internally, even though on the surface (and in the EBNF in appendix A) there is no difference - every value should be an expression.

Odd constructions like the following were used to circumvent problems such as cycles in the grammar:

GroupedExpression $\rightarrow$ LBRACKET Expression RBRACKET

GroupedExpression $\rightarrow$ Value

This causes a few minor annoyances, for example operands to operators having to be parenthesised in order to be parsed, even though the parentheses might seem superfluous.

The grammar was also much longer than the EBNF in appendix A with a total of 49 rules.

Chapter 6

Conclusion

6.1 Lessons Learnt

While bottom-up LR parsers are efficient, they have several drawbacks.

- The parse tables they use are too large and complex to be generated by hand. LR parsers generate many states - there are 93 distinct parse states in Toyscript's grammar.
- Good error reporting is difficult to achieve, as they don't have a lot of contextual information when they encounter a failure.

For these two reasons, this project was very difficult to debug. A failure could mean that there was an error in one or more places in the grammar, or worse still, with the rather opaque table-generating algorithm itself (more than once this was the case).

Other kinds of parsers, like those of the top-down recursive descent variety, are less efficient but do not suffer from these drawbacks. In particular, when a recursive descent parser encounters a failure, it has a greater amount of contextual information about exactly which construct in the grammar was not satisfied.

6.2 Future Work

Parser-generators also have to be robust and featureful, as they are tools in and of themselves. Many conveniences and additional affordances could be made, such as:

- Having a meta-syntax for specifying grammars
- Detecting and reporting errors and ambiguities in grammars
- Better error messages in response to syntax errors
- Conveniences for operator precedence (as opposed to having to encode them as rules)
- Tighter integration with a lexical analysis frontend
- Syntactic sugar for repetition of symbols
- The ability to recognize more classes of grammars

In the short time frame given this project it was difficult to support all these conveniences. These would be areas I would work on if time permitted.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers - principles, techniques, and tools.* Pearson Addison-Wesley.

Knuth, D. E. (1965). *On the translation of languages from left to right* (Technical report). Mathematics Department, California Institute of Technology, Pasadena, California.

Appendix A

Toyscript

Toyscript is an ancillary language created to demonstate the parser frontend developed in this project. It is minimal, with syntax resembling that of OCaml. The grammar may be expressed in EBNF as follows:

```
<expression> ::= <number> | <boolean> | <pair>
               | <id>
               | (<expression>)
               | <unary-op> <expression>
               | <expression> <binary-op> <expression>
               | if <expression> then <expression> else <expression>
               | let <id> = <expression> in <expression>
               | fun ({<id>}) -> <expression>
               | <id>({<expression>})
```

It is well-suited to list processing. Examples of this, along with definitions for higher-order functions such as `map` and `foldr`, may be found in the accompanying code.

A.1 Features

- Fully expression-oriented: constructs in the language may be composed easily

- Pairs as a primary data structure
- 8 levels of operator precedence
- Dynamic typing
- Lexical scope
- Higher-order functions