

Scalabison: A Tool for Recursive Ascent-Descent Parser Generator

Chaya P

Assistant, Professor, Department of Information Science and Engineering, GSSSIETW, Mysuru, Karnataka, India

ABSTRACT

ScalaBison is a parser generator accepting bison syntax and generating a parser in Scala. The generated parser uses the idea of “recursive ascent-descent parsing,” that is, directly encoded generalized left corner parsing. Of interest is that fact that the parser is generated from the LALR (1) tables created by bison, thus enabling extensions such as precedence to be handled implicitly.

Keywords: LALR, Left Corner Parsing, Recursive Ascent-Descent Parsing

I. INTRODUCTION

Recursive ascent-descent parsing was proposed by Horspool [3]. The idea is to combine the power of LR parsing with the small table size and ease of inserting semantic productions available in LL parsing. Furthermore, the generated parsers can be directly encoded, in that the control is handled through executable code rather than indirectly through a table that is then interpreted. In this section we describe these concepts in greater detail.

II. METHODS AND MATERIAL

A. Left-corner parsing

Demers [2] introduced “generalized left corner parsing” which (roughly) combines the benefits of LL and LR parsing techniques. When using top-down or predictive parsing (LL) for parse the yield of a given non-terminal, one requires that the parser identify (“predict”) which production will be used. Left-recursive grammars cannot be used because no bounded amount of look-ahead can determine which production should be chosen. On the other hand,

bottom-up (LR) parsers can follow multiple productions as long as the ambiguity is resolved by the end of the productions. Intuitively, LL parsers require that a decision be made at the start of the productions, whereas LR can wait until the end.

Modern LL parser generators overcome some of the limitations of LL parsing by permitting the grammar writer to include code to help disambiguate cases. This is possible because the top-down parsing technique is intuitive. The disadvantage is that the grammar starts

To accrete implementation details that obscure its clarity. On the contrary, bison, especially with its precedence extensions, enables grammars to be written in a clean and clear style.

The key technique in order to perform left-corner parsing is to determine the recognition points for each production in the grammar, the points where ambiguity is resolved.

Horspool generalizes recognition points into free positions which are points where a semantic action can be inserted. The recognition point is always a free position, but not vice versa since ambiguity may arise after a free position.

B. Recursive ascent-descent parsing

Recursive descent parsing is a well-known implementation technique for predictive parsing. The parser is directly encoded as a set of mutually recursive functions each of which parses a particular non terminal.

Recursive ascent parsing uses recursive functions to directly encode a bottom-up parser. The set of mutually recursive functions consists of one function for each LR parsing state. It seems the concept was invented independently by Roberts [9] and by Kruseman Aretz [4]. Direct encoding can lead to a faster parsing for the same reason that compilation usually leads to faster execution than interpretation. Horspool [3] explains that recursive ascent parsing has not been seen as practical because of the large code size (large tables) and unintuitiveness of the technique.

C. Precedence and other extensions

The bison tool (and its forerunner yacc) includes the ability to declare the precedence and associativity of terminals enabling grammars with operators to have smaller tables. The technique gives a way to resolve shift-reduce and reduce-conflicts without the need to add new states, a new kind of parsing or any sort of indirection. Any remaining parse table conflicts are resolved in a repeatable way. Finally bison includes an error symbol that affects error recovery. Together these extensions change the theoretical nature of the parsing problem. Thus any tool which seeks to duplicate bison's semantics of parsing cannot simply depend on generalized left-corner parsing theory.

D. Architecture of ScalaBison

The key design decision behind ScalaBison was to delegate the table construction to bison. This enables

us to match the syntax and semantics of bison (including its parse table disambiguation techniques) without needing to duplicate the functionality. On the other hand, this decision is limiting in that we cannot create new parsing states arbitrarily – we can only reuse (and adapt!) the ones given to us by bison. Furthermore, it also means our tool is tied to a particular textual representation of parse tables. Fortunately, it seems the format of bison's "output" seems stable.

ScalaBison performs the following tasks:

1. Invoke the bison parser generator;
2. Read in the grammar and generated LALR(1) tables from bison;
3. Determine a recognition point for each production;
4. Identify the set of unambiguous non-terminals: non-terminals occurring after the recognition point of some production;
5. For every unambiguous non-terminal, identify a bison state to adapt into a left-corner (LC) state, and perform the adaptation;
6. Write out the parser boilerplate;
7. Write a function for each terminal (match or error) and unambiguous non-terminal (start a recursive ascent parse at its LC state);
8. Write a function for parsing each production after its recognition point using The previous functions for each symbol;
9. Write a recursive ascent function for each LC state.

In this paper, we pass over most of these tasks without comment. The interesting steps are Step iii and Step v. We briefly note however that the start symbol S will always be in the set of unambiguous non-terminals determined in Step iv because of the artificial production $S' \rightarrow S\$$ added by the generator.

E. Recognition Points

A recognition point is determined by finding the left-most position in each production which is "free" and all following positions are "free." We modify

Algorithm 2 of Purdom and Brown [8] The published algorithm does a computation over a graph for each state and each potential look ahead checking whether each relevant item dominates the action for the given look ahead symbol. We instead use a single graph for each LALR (1) state and check whether items dominate each parse action it can reach.

Precedence and associativity declarations are used by bison to resolve certain shift-reduce conflicts in favor of reductions (rather than shifts). So-called “non-associativity” declarations can even introduce parse errors. Thus with appropriate precedence declarations

$a - b - c$ is parsed as $(a-b)-c$ and $e == f == g$

is a parse error. Normally, the recognition point for binary operators is directly before the operator, but then the recursive descent part of the parser would need to be context-sensitive so that the expression starting with b terminates immediately rather than extending incorrectly through “ $- c$ ” as it would normally. Thus for correctness, we force the recognition point of any rule using precedence to be put at the end.

F. Generating LC States

For each unambiguous non-terminal N , we must by definition have a production $N \rightarrow \alpha N \beta$ in which this non-terminal occurs after the recognition point. We wish to construct a LC state for the item $N \# \rightarrow \alpha \cdot N$ for a new artificial non-terminal $N \#$. This state will be used when (predictively) parsing N . In order to avoid having to create states (and the associated parse actions) ourselves, we find an existing LALR state that contains the item $N \rightarrow \alpha \cdot N \beta$. We use this existing state when defining the parsing actions (after the next paragraph). Figure 1 gives an example of a LALR (1) state being adapted as an LC state. The first item (for rule 7) is irrelevant to the LC state generation and is ignored. We rather use the artificial item show at the extreme right. First we “close” the new LC state, by adding new items $N' \rightarrow \alpha$ for every production $N' \rightarrow \alpha$ whenever an item

with N' immediately after the dot, provided that the recognition point occurs the dot in the item. This last condition distinguishes the process from traditional LR state generation. In Figure 1, no new items are added: the items marked REMOVED occur only in the LALR (1) state, not the LC state.

```

7 class_decl: CLASS TYPEID formals superclass '{'
    feature_list . '}' _: |- feature_list .
12   feature_list: feature_list . feature ';'
13   | feature_list . error ';'
14   | feature_list . NATIVE ';'
15   | feature_list . '{ block }'
16   feature: . opt_override DEF OBJECTID
formals ':' TYPEID '=' expr REMOVED
17   | . opt_override DEF OBJECTID formals ':'
TYPEID NATIVE REMOVED
18   | . VAR OBJECTID ':' TYPEID '=' expr
REMOVED
19   | . VAR OBJECTID ':' NATIVE REMOVED
20   opt_override: . OVERRIDE REMOVED
21   | /* empty */ REMOVED
error    shift, and go to state 49    goto 14
NATIVE   shift, and go to state 50    announce
rule 14
OVERRIDE shift, and go to state 51    announce
rule 12
VAR      shift, and go to state 52    announce rule 12
'{'     shift, and go to state 53    announce rule 15
'}'     shift, and go to state 54    accept feature_list
DEF      reduce using rule 21 (opt_override)
        announce rule 12
feature go to state 55    REMOVED
opt_override go to state 56    REMOVED
default accept feature_list    accept feature_list

```

Figure 1. Adapting an LALR state as an LC state.

The parsing actions of an LC state are adapted from the actions for the basic LALR state. Shift and goto actions lead to (potentially) new LC states after moving the dot over the appropriate symbol, again provided that this does not move the dot past the

recognition point. Otherwise, we need to consider whether an “announce” action is appropriate at this point. In Figure 1, the only shift/goto to remain is the one for error.

Left-corner parsing does not have reduce actions. Instead it has announce actions when the recognition point of a production is reached. If the recognition point is at the far end of a production, the announce action has the same effect as a reduce; otherwise, it means that top-down (predictive) parsing will be used for the remainder of the production.

When adapting a reduce action from the LALR state, the corresponding announce action is used if the relevant item is in the LC state. Otherwise (as seen in the example, without an item for rule 21), we are in a similar situation as we were with a shift action that does not immediately translate. There are two possibilities: one that the action corresponds to an item (or items) of the LALR state that are irrelevant to the LC state that uses it, or that it indicates parsing action beyond the recognition point of an item in the LC state. We detect the latter case by tracing the items (in the LALR state) associated with the action back to where they are generated. If we encounter an item that in the LC state is at its recognition point, we use an announce action. In this way, the reduce action on DEF is converted into an announce of rule 12. Otherwise the action is omitted from the LC state unless an “accept” action is possible (see next paragraph), in which case we use that action (as seen in Fig 1 for ‘}’).

If the LC state contains the artificial item $N\# \rightarrow \vdash N \cdot$ (as in the example,

```
Private def yystate13(yyarg1:Features): YGoto={
  Var yygoto :YYGoto = null;
  yycur match {
```

```

    case          YYCHAR('}')          =>
yygoto=YYNested(YYBase(YYNTfeature_list(yyarg1)
));
    case          OVERRIDE(          )          =>
yygoto=yynest(1,YYNTfeature_list(yyrule12(yyarg1)
))
    case          DEF(          )=>
yygoto=yynest(1,YYNTfeature_list(yyrule12(yyarg1)
))
    case          NATIVE(          )=>
yygoto=yynest(1,YYNTfeature_list(yyrule14(yyarg1)
))
    case          VAR(          )=>
yygoto=yynest(1,YYNTfeature_list(yyrule12(yyarg1)
))
    case          YYCHAR          ('{')=>
yygoto=yynest(1,YYNTfeature_list(yyrule15(yyarg1)
))
    case          _          =>          yygoto
=YYNested(YYBase(YYNTfeature_list(yyarg1)));
  }
  While(true) {
  Yygoto match {
  Case YYNested (g) => return g;
  Case YYBase(YYNTError(s)) =>
  Yyerror(s)
  Yypanic({t:YYToken=> t match {
  Case YYCHAR('.')=>true
  Case _ => false
  }
  }
  )
  Yygoto = yystate14(yyarg1);
  Case x@YYBase(_:YYNTfeature_list) => return x;
  }
  }
  Yygoto
  }

```

Figure 2. Generated Scala code for the LC state from Figure.1.

```

/** Recursive descent parser after recognition point
 * feature_list: feature_list . feature ';'
 */
private def yyrule12(yyarg1 : Features) : Features = {
  var yyresult : Features = null;
  val yyarg2 : Feature = parse_feature();
  parse_YCHAR(';');
  { yyresult = yyarg1 + yyarg2; }
  yyresult
}

```

Figure 3. Recognition function for Rule 12.

where N # is written _ for concision), then we add the default action to “accept” the non-terminal N . Although this adaptation requires some work, by using the LALR states, we avoid the need to propagate look aheads or to negotiate parsing conflicts using precedence rules or other policies.

Figure 2 shows the generated Scala code for the LC state adapted in Figure 1. The yynest and YYNested wrappers are used to help simulate the multi-frame return instruction for recursive ascent parsing. They also catch error exceptions and convert them into error values.

Figure 3 shows the recognition function (predictive parsing routine) for rule 12. This function is called when implementing an “announce” action (as seen in Figure 2). It includes the semantic action code: in this case translated from { \$\$ = \$1 + \$2; }.

The final kind of generated function is the one that starts a recursive descent parse for a given non-terminal. Figure 4 shows the parsing function for the “feature” non-terminal. Such functions are not private so that they can be used by the code that interfaces with the generated parser.

```

def parse_feature() : Feature = {
  yystate17() match {
    case YYBase(YYNTfeature(yy)) => yy
    case YYBase(YYNTerror(s)) => throw new YYError(s)
  }
}

```

Figure 4. Parsing function for the “feature” non terminal

The generated parser has a simple interface to the scanner: the parser is started by passing it an iterator that returns the tokens.

III. RELATED WORK

The number of parser generators using LL or (LA)LR technologies is great. There are fewer tools generating recursive ascent parsers [5,10], and to our knowledge only Horspool has previously written a recursive ascent-descent parser generator.

Parser combinators

The primary mechanism for text parsing included with the Scala standard library is that of parser combinators [6]. Parser combinators are an embedded DSL in Scala for expressing EBNF-like grammars.

At a very high level, parser combinators are a representation of LL(k) parsing without using tables or explicit recursion. Instead, input is consumed by a Parser, which reduces to either Success or Failure, dependent upon whether or not the input was successfully parsed. In general, the combinator use backtracking which impacts efficiency negatively. Grammars of arbitrary complexity may be represented by combining parsers to produce a composite parser, etc.

Figures 5 and 6 compare these two radically different ways of writing parsers. The ScalaBison example is simple and clean for addition and subtraction, but much more verbose than using combinators for

actual parameters. An additional point of comparison is that the ScalaBison parsers are more efficient in time and space .

```
%left '-' '+'
...
expr  : ...
      | expr '+' expr  { $$ = add($1,$3); }
      | expr '-' expr  { $$ = sub($1,$3); }
...
actuals : '(' ')'      { $$ = Nil; }
        | '(' exp_list ')' { $$ = $2; }
        ;
exp_list: expr          { $$ = List($1); }
        | exp_list ',' expr { $$ = $1 + $3; }
        ;
```

Figure 5. Parsing addition/subtraction and actuals with ScalaBison.

```
def exprNoEq = exprNoAdd ~ rep(
  "+" ~> exprNoAdd ^^ { e => add(_:E, e) }
  | "-" ~> exprNoAdd ^^ { e => sub(_:E, e) }
) ^^ collapse
...
def actuals = "(" ~> repsep(expr, ",") <~ ")"
```

Figure 6. Parsing addition/subtraction and actuals with combinators

IV. CONCLUSION

ScalaBison is a practical parser generator for Scala built on recursive ascent-descent technology that accepts bison format input files. It uses bison's LALR(1) tables to build its own LC tables and thus is able to provide the same semantics of conflict resolution that bison does. The parsing speed and space usage are competitive with Scala's built-in parser combinators.

V. REFERENCES

1. Alexander Aiken. Cool: A portable project for teaching compiler construction. ACM SIGPLAN Notices, 31(7):19–26, July 1996.
2. Alan J. Demers. Generalized left corner parsing. In Conference Record of the Fourth ACM Symposium on Principles of Programming

- Languages, Los Angeles, California, USA, pages 170–182. ACM Press, New York, January 1977.
3. R Nigel Horspool. Recursive ascent-descent parsing. Journal of Computer Languages, 18(1), 1993. [4] F. E. J. Kruseman Aretz. On a recursive ascent parser. Information Processing Letters, 29(4):201–206, 1988.
4. Ren'e Leermakers. Non-deterministic recursive ascent parsing. In Proceedings of the fifth conference on European chapter of the Association for Computational Linguistics, pages 63–68. Association for Computational Linguistics, Morristown, NJ, USA, 1991.
5. Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U.Leuven, February 2008.
6. Arthur Nunes-Harwitt. CPS recursive ascent parsing. In ILC 2003, New York City, New York, USA, October 12–15. 2003.
7. Paul Purdom and Cynthia A. Brown. Semantics routines and LR(k) parsers. Acta Informatica, 14:299–315, 1980.
8. G H. Roberts. Recursive ascent: an LR analog to recursive descent. ACM SIGPLAN Notices, 23(8):23–29, 1988.
9. Ronald Veldema. Jade, a recursive ascent LALR(1) parser generator. Technical report, Vrije Universiteit Amsterdam, Netherlands, 2001.