Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 253 (2010) 65–74

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

TOOL PAPER: ScalaBison Recursive Ascent-Descent Parser Generator [1]

John Boyland[2] Daniel Spiewak[2]

Department of EE & Computer Science
University of Wisconsin-Milwaukee
Milwaukee, Wisconsin, USA

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: Parsing, recursive ascent-descent parsing, parser generators, scala

1 Introduction

Recursive ascent-descent parsing was proposed by Horspool [4]. The idea is to combine the power of LR parsing with the small table size and ease of inserting semantic actions 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.

1.1 Left-corner parsing

Demers [3] introduced "generalized left corner parsing" which (roughly) combines the benefits of LL and LR parsing techniques. When using top-down or predictive parsing (LL) to parse the yield for a given nonterminal, one requires that the parser identify ("predict") which production will be used. Left-recursive grammars cannot

[1] Work supported in part by the National Science Foundation (CCF-0702635). The opinions expressed here are not necessarily those of the National Science Foundation or the US Government.

[2] Email: {boyland,dspiewak}@uwm.edu

doi:10.1016/j.entcs.2010.08.032

be used in general because unbounded lookahead may be required to 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 time we reach the end of any production. Intuitively, LL parsers require that a decision be made at the start of the productions, whereas LR can wait until the end. Thus, LR theory permits a greater number of grammars to be parsed deterministically.

The greater power of LR is offset by the greater complexity of the parser, by the larger tables generated and by the limitation of where semantic actions can occur in the grammar. The last item is somewhat of a red herring, because modern parser generators such as bison based on LR(1) (or its simplification LALR(1)) permit semantic actions to appear just about anywhere an LL(1) parser generator would, assuming the grammar is indeed LL(1).[3] The larger size of tables is less of a problem for today's computers, especially when compression techniques are used. However, the greater complexity of the parser means it is much harder for the user of the parser generator to understand what is happening.

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 declarative style.

The intuition behind generalized left-corner parsing is that during LR parsing, few productions must wait until the end to resolve ambiguity. Frequently, the production that will be used is identified long before its end. Thus in left-corner parsing, the parser switches from bottom-up to top-down parsing as soon as the correct production is identified. This has two benefits over straight LR parsing: the tables are smaller and prediction makes it easier to generate useful error messages—if one terminal is predicted, it is easy to indicate the expectation in an error message in case a different terminal is encountered.

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 in some unusual cases [4], non-free positions occur *after* an earlier free position. In this paper, we choose the earliest free position that has no following non-free positions as the recognition point.

1.2 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 nonterminal.

Recursive *ascent* parsing uses recursive functions to directly encode a bottom-

[3] "Just about" because there are a few unusual cases where this correspondence does not hold.

up parser. The set of mutually recursive functions consists of one function for each LR parsing state. Pennello [8] gives an assembly language implementation of a directly encoded LR parser. It seems the concept was later invented independently by Roberts [10] and by Kruseman Aretz [5]. Direct encoding can lead to a faster parsing for the same reason that compilation usually leads to faster execution than interpretation. Horspool [4] explains that recursive ascent parsing has not been seen as practical because of the large code size (large tables) and unintuitiveness of the technique. Recursive ascent parsers would be too tedious to write by hand, and the generated parsers are not hospitable to human injection of semantic routines.

Generalized left-corner parsing's advantages *vis-a-vis* LR parsing are directly relevant: they lead to smaller tables and after disambiguation, using top-down parsing in which it is easy to place semantic actions. Horspool [4] showed that the advantages are real—parsing can be almost three times faster as opposed to with yacc, and still enable hand-editing of semantic actions.

1.3 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. Any remaining parse table conflicts are resolved in a repeatable way. (Neither kind of resolution is always benign—the resulting parser may reject strings that can be generated by the grammar.) 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 use generalized left-corner parsing theory.

2 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, the format of bison's "output" seems stable. We have been able to use bison version 1.875 as well as 2.3.

ScalaBison performs the following tasks:

(i) Invoke the bison parser generator;

(ii) Read in the grammar and generated LALR(1) tables from bison;

(iii) Determine a recognition point for each production;

(iv) Identify the set of *unambiguous nonterminals*: non-terminals occurring after the recognition point of some production;

(v) For every unambiguous nonterminal, identify a bison state to adapt into a left-corner (LC) state, and perform the adaptation;

(vi) Write out the parser boilerplate;

(vii) Write a function for each terminal (match or error) and unambiguous nonterminal (start a recursive ascent parse at its LC state);

(viii) Write a function for parsing each production after its recognition point using the previous functions for each symbol;

(ix) 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 nonterminals determined in Step iv because of the artificial production $S' \rightarrow S\$$ added by the generator.

2.1 Recognition Points

The recognition point for a production is determined by finding the left-most position in each production which is free and for which all following positions are free. Recall that a "free" position is one in which a semantic action can be inserted without introducing a parse conflict. At worst, the recognition point is after the end of the production.

We modify Algorithm 2 of Purdom and Brown [9] to determine free positions. The published algorithm does a computation over a graph for each state and each potential lookahead checking whether each relevant item dominates the action for the given lookahead symbol. We instead use a single graph for each LALR(1) state. We check for each item whether, in this graph, it dominates each parse action it can reach. If it doesn't, this means that at the point where the parse has progressed to this item, there is still ambiguity, and thus the item is not free.

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 production using precedence to be put at the end. This safety measure is required whenever a shift-reduce conflict is resolved in favor of a reduce (which in bison only happens with precedence).

```
 7 class_decl: CLASS TYPEID formals superclass '{' 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
17        | . opt_override DEF OBJECTID formals ':' TYPEID NATIVE
18        | . VAR OBJECTID ':' TYPEID '=' expr
19        | . VAR OBJECTID ':' NATIVE
20 opt_override: . OVERRIDE
21             | . /* empty */

error     shift, and go to state 49
NATIVE    shift, and go to state 50
OVERRIDE  shift, and go to state 51
VAR       shift, and go to state 52
'{'       shift, and go to state 53
'}'       shift, and go to state 54

DEF  reduce using rule 21 (opt_override)

feature       go to state 55
opt_override  go to state 56
```

Fig. 1. An (augmented) LALR state generated by bison.

2.2 Generating LC States

An LC parser uses LR parsing techniques until it has determined which production to use, as determined when it reaches the recognition point. At this point, the production is "announced" and the remainder of the production is parsed using LL techniques. At latest, a production is announced at the point where the LR parser would reduce it (at the end). Thus an LC parser has no reduce actions but rather "announce" actions.

Once a production is announced, the parser predicts each remaining symbol in turn. Terminals are handled by simply checking that the next input symbol matches. Nonterminals are handled by calling a routine specially generated to parse this nonterminal. Here, we revert back to LR-style parsing, and thus we need a parse state to parse this nonterminal. Following Horspool, we generate this parse state for parsing N around the core consisting of a single item $N^\sharp \rightarrow \ \vdash \ \cdot N$ for a new artificial nonterminal $N^\sharp$. The $\vdash$ is used to ensure that the item will be be seen as "core."

A similar artificial nonterminal and item is used in LR parser generation for the start state (alone); when the end of this production is reached, the parser considers an "accept" action. For an LC parser, "accept" actions are possible for any (unambiguous) nonterminal.

In order to avoid having to determine the parse actions ourselves, we find an existing LALR state that contains the item $N_0 \rightarrow \alpha \cdot N\beta$. We then adapt the LALR state's actions (see below) to get the LC state's actions. In the process of creating an LC state, we may need to create new states to receive shift/goto actions. This process continues until no new LC states must be created. Then we move on to the next nonterminal that needs its own parse state (each "unambiguous" nonterminal needs one) until all are handled.

Figure 1 shows an example LALR state whose actions are adapted for the LC state shown in Fig. 2. For the LC state, we start with the five items shown. Then we "close" the new LC state, by adding new items $N' \rightarrow \cdot\alpha$ for every production

```
   feature_list# : |- feature_list .
12 feature_list: feature_list . feature ';'
13             | feature_list . error ';'
14             | feature_list . NATIVE ';'
15             | feature_list . '{' block '}'

error          go to state 14
NATIVE         announce rule 14
OVERRIDE       announce rule 12
VAR            announce rule 12
'{'            announce rule 15
'}'            accept feature_list
DEF            announce rule 12
$default       accept feature_list
```

Fig. 2. The LC state formed by adapting the LALR state in Fig. 1.

$N' \to \alpha$ whenever the LC includes an item with N' immediately after the dot, *provided* that the recognition point occurs after the dot in the item. This last condition distinguishes the process from traditional LR state generation. In Fig. 2, no new items are added because the items for rules (productions) 12, 14 and 15 are all at their recognition point, and the item for rule 13 has the artificial nonterminal "error" after the dot.

Shift 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, for a shift action, we need to determine what "announce" action is appropriate at this point (see below). In Fig. 2, the only shift/goto to remain is the one for `error`.

When adapting a reduce action from the LALR state, we also need to determine what announce action is appropriate—it is not always the one the LALR state was going to reduce, because the corresponding item might not be in the LC state. Thus, both for shift actions that go past the recognition point (such as on `OVERRIDE`) and for reduce actions (such as on `DEF`), we need to determine whether an announce action should be done instead. We do this by tracing the item corresponding to the action back to find how it was added to the LALR state.

For `OVERRIDE`, we trace the shift action to the item on rule 20 which came about during closure of the LALR state from items for rules 16 and 17 which in turn came from closure on the item for rule 12. This last item is in the LC state and thus we use the "announce rule 12" action for this input. The shift action on `VAR` gives the same conclusion. The shift actions on `NATIVE` and `'{'` are mapped to "announce rule 14" and "announce rule 15" actions respectively, through simpler applications of this process. The shift action for `'}'` leads to a different outcome. When we trace it back we get to the item for rule 7, which is a core item of LALR state, but absent in the LC state. Thus no announce action is generated for `'}'`. We return to this case below. For the reduce action on `DEF`, we trace the action back to rules 16

```
private def yystate13(yyarg1: Features) : Int = {
  var yygoto : Int = 0;
  try {
  yycur match {
    case YYCHAR('}') => yygoto = 2; yynt = YYNTfeature_list(yyarg1);
    case NATIVE() => yygoto = 1; yynt = YYNTfeature_list(yyrule14(yyarg1))
    case YYCHAR('{') => yygoto = 1; yynt = YYNTfeature_list(yyrule15(yyarg1))
    case DEF() => yygoto = 1; yynt = YYNTfeature_list(yyrule12(yyarg1))
    case OVERRIDE() => yygoto = 1; yynt = YYNTfeature_list(yyrule12(yyarg1))
    case VAR() => yygoto = 1; yynt = YYNTfeature_list(yyrule12(yyarg1))
    case _ =>  yygoto = 2; yynt = YYNTfeature_list(yyarg1);
  }
  } catch {
    case YYError(s) => yynt = YYNTerror(s);
  }
  while (yygoto == 0) {
    yynt match {
      case YYNTerror(s) =>
        yyerror(s)
        yypanic({ t:YYToken => t match {
          case YYCHAR(';') => true
          case _ => false
        }})
        yygoto = yystate14(yyarg1);
      case _:YYNTfeature_list => return 0;
    }
  }
  yygoto-1
}
```

Fig. 3. Generated Scala code for the LC state from Fig.2.

```
/** 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_YYCHAR(';');
  { yyresult = yyarg1 + yyarg2; }
  yyresult
}
```

Fig. 4. Recognition function for Rule 12.

and 17, and thus back to rule 12 and thus generate the “announce rule 12” action for DEF.

If the LC state contains the artificial item $N^\sharp \to \vdash N\cdot$ (as in the example, where N is `feature_list`), then we add the default action to “accept” the nonterminal N. This default action is also used for any actions left undefined previously (as with the action for `'}'`).

Although this adaptation requires some work, by using bison’s LALR states, we preserve bison’s resolution semantics, while avoiding the need to propagate lookaheads or to negotiate parsing conflicts using precedence rules or other policies.

Figure 3 shows the generated Scala code for the LC state in Fig. 2. The `try` block is used to handle parse errors (because the state can handle the `error` pseudo-nonterminal). We simulate multiple-level returns for the recursive ascent parse functions by putting the return value in field `yynt` and returning the number of frames (`yygoto`) that must still be popped.

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

Fig. 5. Sample parse routine for a nonterminal.

Figure 4 shows the recognition function (predictive parsing routine) for rule 12. This function is called when implementing an "announce" action (as seen in Fig. 3). It includes the semantic action: 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 nonterminal. Figure 5 shows the parsing function for the "feature" nonterminal. This routine is called from the code in Fig. 4. Such functions are not private so that they can be used by the code that interfaces with the generated parser.

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

3 Related Work

The number of parser generators using LL or (LA)LR technologies is great. There are fewer tools generating recursive ascent parsers [6,2,11], and to our knowledge only Horspool has previously written a recursive ascent-descent parser generator.

The primary mechanism for text parsing included with the Scala standard library is that of parser combinators [7]. Parser combinators are an embedded DSL in Scala for expressing EBNF-like grammars. The executable code is generated directly by the Scala compiler, there is no need for an external tool (such as ScalaBison) to process the grammar description. At a very high level, parser combinators are a representation of LL(*) parsing without using tables. 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, combinators use backtracking which impacts efficiency negatively. Grammars of arbitrary complexity may be represented by composing smaller parsers.

4 Evaluation

One common concern with recursive ascent parsers is that the large number of states leads to a large code size. Indeed Veldema [11] decided against a purely direct-encoded parser for this very reason, opting instead for an approach that can be seen as table-driven. On the other hand, Bhamidipaty and Proebsting [2] found that a directly encoded (recursive ascent parser) using `goto` statements instead of recursive routines yields parsers only twice the size of their `yacc` counterparts. Recursive

	Compiled	`good.cl`		`large.cl`	
Generator	Size (K)	Time (ms.)	Space (MB)	Time (ms.)	Space (MB)
combinators	350	54	3.1	275	3.3
ScalaBison	200	17	0.2	36	2.1
Beaver	70	8	0.2	19	1.5

Table 1
Comparing ScalaBison with other generators.

ascent-descent parsing is supposed to alleviate the size problem and indeed we find that the number of states is noticeably fewer: about 40% fewer for grammars that make heavy use of LR features. For example, the published LALR(1) grammar of Java 1.1 (in which optional elements are expanded to split one production into two, resulting in a grammar with 350 productions) yields 621 states with bison but only 378 in ScalaBison. We also tested a grammar for a dialect of Cool [1] which made heavy use of precedence declarations (67 productions): 149 states for bison, 100 for ScalaBison. The reduction in states is to be welcomed but may not be great enough to make recursive ascent-descent attractive to people repelled by recursive ascent. The generated parser for Cool is still over 100K bytes of Scala, and for Java 1.1 over 600K bytes. By way of comparison, the bison generated parsers are 53K and 120K of C source respectively; bison does a good job compressing the tables and directly encoded parsers don't lend themselves as easily to compression.

To measure performance, we compare the ScalaBison Cool parser with one written using parser combinators. The comparison is not "fair" in that parser combinators were designed for clarity, not speed, and furthermore, the ScalaBison parser uses a hand-written (but simple and unoptimized) scanner whereas the combinator parser operates directly on the character stream. We also compared ScalaBison with Beaver (`beaver.sourceforge.net`), reported to generate the fastest LALR(1) JVM-based parsers.

Table 1 shows the results of testing Cool parsers implemented by all three generators against an Cool input file (`good.cl`) comprised of roughly 3,100 tokens exercising every production of the grammar. The file `large.cl` simply repeats this file ten times. The first column shows the compiled code size. The "Space" columns show the maximum memory usage ("high water mark") during the runs. All tests were performed using a MacBook Pro, 2.4 Ghz Core 2 Duo with 4 GB of DDR3 memory using Apple's JDK 1.5.0_16 and Scala 2.7.3.final. Each test was run twelve times with the best and worst results dropped, and remaining ten times averaged. Garbage collection was triggered between each test.

Beaver generates noticeably faster code. Part of the difference is due to the fact that the numbers for ScalaBison include running the scanner which takes roughly half the reported time, whereas the (different) scanner uses up only a third of Beaver's much smaller time. However, even taking the scanner time out of the parse time still leaves Beaver's parser faster. One factor is that ScalaBison uses `match` clauses (see Figure 3) which the Scala compiler implements with a linear search, whereas Beaver uses (constant-time) array lookup.

5 Conclusion

ScalaBison is a practical parser generator for Scala built on recursive ascent-descent technology that accepts bison format input files. This enables the benefits of direct-encoding while reducing code size from a pure recursive-ascent solution. 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 parsers generated by ScalaBison use more informative error messages than those generated by bison. The parsing speed and space usage are much better Scala's built-in parser combinators but are somewhat slower than the fastest JVM-based parser generators.
SDG

References

[1] Alexander Aiken. Cool: A portable project for teaching compiler construction. *ACM SIGPLAN Notices*, 31(7):19–26, July 1996.

[2] Achyutram Bhamidipaty and Todd A. Proebsting. Very fast YACC-compatible parsers (for very little effort). *Software Practice and Experience*, 2(28):181–190, 1999.

[3] 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.

[4] R. Nigel Horspool. Recursive ascent-descent parsing. *Journal of Computer Languages*, 18(1), 1993.

[5] F. E. J. Kruseman Aretz. On a recursive ascent parser. *Information Processing Letters*, 29(4):201–206, 1988.

[6] René 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.

[7] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U.Leuven, February 2008.

[8] Thomas J. Pennello. Very fast LR parsing. *ACM SIGPLAN Notices*, 21(7):145–151, 1986.

[9] Paul Purdom and Cynthia A. Brown. Semantics routines and $LR(k)$ parsers. *Acta Informatica*, 14:299–315, 1980.

[10] G. H. Roberts. Recursive ascent: an LR analog to recursive descent. *ACM SIGPLAN Notices*, 23(8):23–29, 1988.

[11] Ronald Veldema. Jade, a recursive ascent LALR(1) parser generator. Technical report, Vrije Universiteit Amsterdam, Netherlands, 2001.