RECURSIVE ASCENT-DESCENT PARSERS

R. Nigel Horspool
Department of Computer Science
University of Victoria, P.O. Box 1700
Victoria, BC, Canada V8W 2Y2

Abstract

Non-backtracking recursive descent parsers are easy to create but suffer from the disadvantage that they are limited to the relatively small LL(k) class of grammars. Recursive ascent parsers can be built for the much larger LR(k) class of grammars, but they are likely be too large to be useful and manual insertion of semantic action code into the parsers would require considerable skill. A composite approach, dubbed 'recursive ascent-descent parsing' or XLC(1) parsing, and which is related to left-corner parsing, solves all the problems. The parser has two parts – a top-down part and a bottom-up part. Either or both parts may be implemented as a collection of recursive functions or by a table-driven parsing algorithm. When a table-driven implementation of the bottom-up part is used, any syntax error recovery scheme applicable to LR parsers can be employed. Small, fast, implementations of the parsers are possible.

1 Introduction

Recursive descent, the method by which a LL(1) grammar can be converted into a set of functions that implement a top-down parser, is a well known technique[1,6]. Recently, there have been papers showing how each state in a LR recognizer can be converted to a function, thus implementing a parser for a larger class of grammars[2,8,3,9]. The name recursive ascent parser has been coined to describe such a parser.
Although the larger class of acceptable grammars may seem like a good reason to prefer a recursive ascent over recursive descent, there are strong contrary arguments. One difficulty is that a recursive ascent parser is likely to be composed of many more functions than the equivalent recursive descent parser. A second difficulty is that many users of recursive descent parsers like the freedom to be able to insert semantic action code into the parser by hand. However, it would not be at all easy to deduce where semantic actions should be located in the code of a recursive ascent parser. Inserting a semantic action into a LR(1) grammar can easily cause that grammar to lose the LR(1) property. Similarly, inserting a semantic action into a LALR(1) grammar may cause a loss of the

LALR(1) property. And, in general, the same semantic action code may need to be duplicated in several functions.

Inspired by Demers' work on left-corner parsing[5], we can solve both problems and attain the best of all possible worlds. We can construct a parser which consists of two components. One component performs as much parsing as possible using a top-down method. The other component uses a bottom-up method based on LR parsing algorithms for the remaining parsing actions.

Our approach is more general than left-corner parsing, although there is a strong similarity. It should be noted that either component of the parser may be implemented as a table-driven algorithm or as a directly executed parser. When both components are implemented as directly executable code in the form of recursive descent functions for the top-down component and recursive ascent functions for the bottom-up component, we call the parser a *recursive ascent-descent parser.* When both components are table-driven, our parsing method is a generalization of the *generalized left-corner* parsing method invented by Demers[5]. Since Demers called his method GLC, we might call ours Extended GLC, or XLC(k) for short. This paper is oriented towards the directly executable, recursive ascent-descent, form of parser.

Although we can give general algorithms for building recursive ascent-descent parsers for LR(k) grammars, it would be a good idea to restrict our attention to the case when k is one. The size of the parser for k greater than one is likely to be prohibitive.

If the grammar actually belongs to the LL(1) class, the bottom-up component of the parser will have a trivial structure while the top-down component will be identical, except in some details, to a recursive descent parser. If the grammar is LR(1) and not LL(1), the parser will normally still contain a significant top-down component. Such a parser should contain far fewer functions than the equivalent recursive ascent parser. Furthermore, it will be absolutely clear to the compiler writer exactly where semantic action code may be inserted. Just as with a recursive descent parser, semantic actions may be freely inserted anywhere in the top-down component of the parser.

2 Overview of Recursive Ascent-Descent Parsing

A bottom-up parser works by recognizing right hand sides of production rules and reducing these right hand sides to the corresponding lefthand side symbols. When the parser begins to recognize symbols from a right hand side, there may not be a unique choice as to which production rule is involved. For example, a grammar for the Modula-2 language might contain two production rules similar to the following.

$$\begin{array}{lcl} \text{Statement} & \rightarrow & \textbf{if}\ \text{expression}\ \textbf{then}\ \text{statement-list}\ \textbf{fi} \\ \text{Statement} & \rightarrow & \textbf{if}\ \text{expression}\ \textbf{then}\ \text{statement-list}\ \textbf{else}\ \text{statement-list}\ \textbf{fi} \end{array}$$

When the parser encounters the **if** keyword, it can narrow down the possible production rules to a choice between these two. However, the choice does not become unambiguous until either the **else** keyword or the **fi** keyword is encountered. We will refer to this ambiguity between possible right hand sides as *rule ambiguity.* It should not be confused with the notion of grammatical ambiguity when two distinct derivations for the same

sentence exist. The presence of rule ambiguity is not purely a property of the grammar, it also depends on the parsing algorithm in use.
It is quite possible that while matching consecutive symbols in a right hand side, a rule ambiguity disappears and then a new rule ambiguity is created. A small grammar that illustrates this possibility is the following.

1.	A	$\rightarrow$	a B b c
2.	B	$\rightarrow$	B b
3.	B	$\rightarrow$	d

Consider the first production rule when a LR(1) or LALR(1) parsing algorithm is used to parse the input `a d b c`. Initially, when the symbol `a` is encountered, the choice of rules is unambiguous. It is still unambiguous after the `a` has been shifted, and `d` is the current symbol. But after the `d`, there is an ambiguity as to whether that `b` belongs to the right hand side of rule 1 or to the right hand side of rule 2. Thus, between the `B` and `b` symbols on the right hand side of rule 1, a rule ambiguity exists when a parsing method with only one symbol of lookahead is used. However, the rule ambiguity is resolved as soon as another symbol is read and `c` becomes the input symbol.
Our notion of rule ambiguity is exactly equivalent to the concept of *free positions* due to Purdom and Brown[7]. Purdom and Brown characterize a free position in the right hand side of a rule as being a position where a new null symbol (perhaps employed as a semantic action marker) may be freely inserted without affecting the grammar class. A null symbol is a non-terminal symbol whose only definition is a production rule with an empty right hand side. Intuitively, a semantic action may be associated with a particular position in a rule only if it is unambigous at that point in the parse as to which rule is being recognized. And this is, of course, the same as rule ambiguity.
The key idea of our hybrid parsing algorithm is that top-down techniques may be used for parsing symbols in a rule's right hand side at each of the rule's free positions. If (and only if) the grammar is LL(1), every position in every right hand side is free. In general, however, there will be some positions that are not free. Suppose, for example, that only the three positions marked by an underscore character in the following rule are free.

```
A  →  A B _ C D E _ F G _
```

If we wish to create a recursive descent function to recognize the right hand side of this production rule, the function should have the following form:

```
procedure A1() {
     call C_D_E();
     call F_G();
}
```

This function, $A1$, would be invoked by the bottom-up component of the parser after the A and B symbols have been recognized. Prior to this point, a rule ambiguity would have existed. $A1$ calls the bottom-up component of the parser to match the C, D and E symbols. They must necessarily be matched as a single unit because of the existence

of rule ambiguity between the C and D and between the D and E. After control returns from C_D_E, $A1$ again invokes the bottom-up component of the parser to match the F and G symbols as a single unit.
In practice, it is unusual for a non-free position to occur to the right of a free position in a production rule. Thus the common case is that the bottom-up parser is re-invoked only to match single non-terminal symbols, just as in left-corner parsing.

2.1 A Simple Example

As a small example, we use the following grammar that was given by Purdom and Brown[7]. Free positions are indicated in the grammar by an underscore character.

```
0.    S → _ T _
1.    T → T _ * _ F _
2.    T → _ F _
3.    F → id _
4.    F → id _ [ _ T _ ] _
```

The corresponding recursive ascent-descent parser, coded using a small extension to C, is shown in Figure 1. Our C extension occurs with **return** statements. We have used the notation **return***k to indicate that a k-level function return is to be made. That is, **return***1 is identical to the normal C **return** statement and simply returns control to the caller of the current function; **return***2 means that control is to be returned to the caller of the caller, and so on. Finally, **return***0 is to be interpreted as a null statement. We leave emulation of the **return***k construct in languages that lack this operation as a simple exercise for the reader. Where no **return** statement appears in a function, we assume that a conventional one-level return is executed when control reaches the end of the function.
Except for some renaming of functions and variables to improve human readability and for the use of the **return***k construct, the program of Figure 1 is as created by our implementation of a recursive ascent-descent parser generator. (In the actual implementation, all names used in the parser are prefixed by the letters 'yy' or 'YY' to reduce the chance of collision with semantic action code supplied by the user.)
The example grammar is so simple that all non-terminal transitions (i.e. goto actions) in the bottom-up parser are uniquely determined. In general, however, the variable *lhs* needs to be tested in the bottom-up component of the parser to select the appropriate non-terminal transition. An optimization discovered by Roberts[8,9] can be used to simplify and to improve the efficiency of the logic that chooses the non-terminal transition. This optimization has been used, for example, in function $Q1$ where there is a perpetual loop calling state $Q3$. (The loop is eventually terminated when $Q3$ performs a multilevel return.)

3 Constructing Recursive Ascent-Descent Parsers

The algorithm used for the bottom-up component of the parser can be any of the standard LR methods, including SLR(k) , LALR(k) and LR(k) . We will assume that the

```
#define id  257     /* token codes */
#define S   258
#define T   259
#define F   260

#define SCAN()  t = yylex()
#define ERR()  yyerror("bad syntax!")
#define MATCH(x) if(t!=x)ERR();SCAN()

int t;    /* current symbol  */
int lhs;  /* latest LHS */

/*** bottom-up parser component ***/

yyparse() {   /* main entry point */
    SCAN();    /* get first token */
    Q0();              /* match S */
    if (t != EOF) ERR();
    return 0;     /* success code */
}

Q0() {          /*  recognizes S  */
    if ( t == id )
        R0();
    else
        ERR();
}

Q1() {          /* recognizes T */
    if ( t == id )
        R2();
    else
        ERR();
    for( ; ; )
        Q3();
}

Q2() {          /* recognizes F */
    if ( t == id ) {
        SCAN();
        Q4();
    } else
        ERR();
}
```

```
Q3() {
    if ( t == '*' ) {
        R1();
        lhs = T;  return*1;
    } else
        return*2;
}

Q4() {
    switch( t ) {
        case '[':
            R4();  break;
        case EOF: case '*': case ']':
            R3();  break;
        default:   ERR();
    }
    lhs = F;
    return*1;
}

/*** top-down parser component ***/

R0() {          /* rule: S ::= T */
    Q1();             /* match T */
}

R1() {       /* rule  T ::= T * F */
    SCAN();        /* accept '*' */
    Q2();          /* match  F   */
}

R2() {          /* rule: T ::= F */
    Q2();             /* match F */
}

R3() {          /* rule: F := id */
}

R4() {    /* rule: F := id [ T ] */
    SCAN();        /* accept '[' */
    Q1();          /* match  T   */
    MATCH(']');    /* match  ']' */
}
```

Figure 1: Example Recursive Ascent-Descent Parser

LALR(1) method is to be used. (Thus, the resulting parser will be an implementation of the LAXLC(1) parsing method.)
Suppose that we are given a grammar G for which we need to construct a recursive ascent-descent parser. An overview of the construction process is as follows.

1. Construct the sets of items that correspond to the states of the LALR(1) parser for G. If any LALR(1) conflicts are detected, the grammar must be rejected.

2. Apply the Purdom and Brown algorithm to the sets of items and compute the free positions in the production rules of G.

3. Use the free position information and knowledge of lookaheads in the LALR(1) parser to create the core states of the new parser and to create the top-down component of the parser.

4. Perform closure operations to complete the core states and to create additional states in the bottom-up component of the parser.

Books on compiler construction[1,6] or LR parsing[4] may be consulted for full details of step one. The full algorithm for step two is given by Purdom and Brown[7].
After the second step, we will know all the free positions. We will also know the set of valid lookahead symbols for every rule reduction in the LALR(1) parser. For example, suppose that the grammar includes the following rule,

```
A  →  W _ X _ Y Z _
```

where underscore characters mark the free positions. Further suppose that the only reduce items for this rule in the LALR(1) parser are

$$[\mathrm{A} \rightarrow \mathrm{W\ X\ Y\ Z} \bullet\ ;\ \{ab\}\,] \qquad \text{and} \qquad [\mathrm{A} \rightarrow \mathrm{W\ X\ Y\ Z} \bullet\ ;\ \{bcd\}\,]$$

The set {a b} represents the lookahead set associated with the first item, and similarly for the second item. We would then compute the union of the two context sets, namely {a b c d}, as the lookahead set for the rule. Our top-down function that implements the rule must perform the two actions:

match X; *match* Y Z ;

If a single symbol being matched is a terminal symbol, we can we can create code to test the current input and to read a new symbol. In the special case when a terminal symbol is the first symbol to matched in the function, we can omit the test (the bottom-up parsing component will ensure that the current input symbol is valid). But for non-terminal symbols and for groups of symbols (which will always begin with a non-terminal symbol), we must create code that invokes the bottom-up parser.
Assuming that X in our example rule is a non-terminal symbol and that the lookahead set for the rule is σ, we will need to create a LALR(1) -like parser that uses the item

$$[\rightarrow \vdash\ \bullet\ \mathrm{X};\ \mathrm{first}(\mathrm{YZ}\sigma)\,]$$

as the basis of its start state.[1] The symbol $\vdash$ represents a fictitious start-of-input symbol and is present only to maintain the property that no core item has its dot marker in the leftmost position. Similarly, we will need to create another LALR(1) -like parser that has the item

$$[\rightarrow \vdash \ \bullet \mathrm{Y\ Z};\ \mathrm{first}(\sigma)]$$

as the basis of its start state. The body of the function for matching the production rule would simply consist of consecutive calls to these two parsers.

After step three, we will have created the basis items for several start states. One additional start state must be created. If the goal symbol of the grammar is S, we must create a start state with the basis item:

$$[\rightarrow \vdash \ \bullet \mathrm{S};\ \{\dashv\}]$$

where $\dashv$ represents an end-of-input symbol. Step 4 of the algorithm applies a variant of the usual LALR(1) closure process – adding completion items to states and creating new states by computing transitions. While it would be possible to create completely separate parsers from each start state configuration, that would be wasteful. There is no reason why a single parser with several start states cannot be created.

The principal difference between our closure process and the standard LALR(1) closure is as follows. If an item has its dot marker in a position that is free, *no completion or transition items are created from this item.* The effect of ignoring items when transitions are considered causes far fewer parser states to be generated than for the full LALR(1) parser. The special form of rule used in the basis items of start states is assumed to have no free positions in its right hand side.

The code for the bottom-up component of the recursive ascent-descent parser is created from the LALR(1) sets of items. If a state contains the item

$$[\mathrm{A} \rightarrow \alpha \bullet \mathrm{X}\beta;\ \sigma]$$

and the dot marker is *not* at a free position in the rule, the state will have a shift transition (if X is a terminal) or a goto transition (if X is a non-terminal) to some state on symbol X. In a recursive ascent implementation of the bottom-up component of the parser, the shift or goto transition is implemented as a function call. If control returns (it may not return because of the use of multilevel returns in the program), a goto transition must be selected as the next action. A global variable, *lhs* in Figure 1, contains the non-terminal symbol that determines the appropriate goto.

If a state contains the item

$$[\mathrm{A} \rightarrow \alpha \bullet \beta;\ \sigma]$$

and if the dot marker is at a free position, the state will pass control to the top-down component of the parser to complete recognition of the rule $A \rightarrow \alpha \bullet \beta$ whenever the current symbol is a member of the set first($\beta\sigma$). When control returns from the top-down

[1] The *first* function generates the set of starter symbols for its argument, and is defined in the standard way[4].

component, a global variable (*lhs*) is set to the left hand side symbol of the production rule and the statement **return***k must be executed, where k is the position of the dot marker in the item. If the dot occupies the leftmost position in the rule, there are zero levels to return through. This means that a goto transition in the current state, selected by *lhs*, must be executed.
Since every rule of the original grammar contains at least one free position (the rightmost position in a right hand side is guaranteed to be free), the bottom-up component of the XLC(1) parser never performs a reduce action. That action is always performed in the top-down component.
The only other form of item which may occur is an accept item:

$$[\rightarrow \alpha \bullet ; \sigma]$$

It corresponds to successful recognition of the sequence of symbols α. When the current symbol is an element of σ, the correct action is to execute **return***k where $k = |\alpha|$. This action returns control to the caller of the corresponding start state. If this item is the only item in the state, we can coalesce the accept action with the goto transition in the predecessor state, and execute a **return***$(k-1)$ action in that predecessor state instead. This optimization was performed on the parser shown in Figure 1.

4 Generalizing to LR(1) Grammars

The process of merging lookahead sets for rule reductions does lose information that may be required for handling a LR(1) grammar. One solution is to create a different recursive function for each possible lookahead set associated with a production in the top-down component of the parser. For example, if the production rule A $\rightarrow$ W _ X _ Y Z _ (with free positions marked by underscores) has σ as one of its possible lookahead sets, we would create a function with the form

```
A_i() { X_i(); Y_Z_i(); }
```

where X_*i* is a start state in the parser created from the basis item

$$[\rightarrow\vdash \ \bullet \ \mathrm{A};\ \mathrm{first}(\mathrm{YZ}\sigma)]$$

and so on.
An alternative approach, which avoids having several versions of the code for each production rule, is to parameterize the code with the various lookahead sets. The single function generated for the rule, above, would then have one of the two forms:

`A(`s_1`) { X(`s_2`); Y_Z(`s_1`); }` or `A(`s_1`) { X(`$s_1 \cup s_2$`); Y_Z(`s_1`); }`

In both forms, the notation s_1 represents the lookahead set for the rule, and s_2 represents the set first(YZ). The second form is used if both Y and Z are nullable symbols (i.e. if a derivation YZ $\Rightarrow^* \epsilon$ exists); otherwise the first form is used. Of course, since all the lookahead sets are known in advance, there is no need to dynamically compute set unions or to pass sets to functions. A simple numbering scheme for the distinct sets that occur in the parser may be employed instead.

5 Error Recovery?

If the bottom-up component of the XLC(1) parser is table-driven and if the top-down component keeps track of how many symbols in each active right-hand side have been matched, we have all the information needed to implement full LR recovery after a syntax error. From this information, it would be possible to construct an analogue to the usual LR state stack. Given the LR state stack, it would be possible, in principle, to implement any desired recovery strategy.
However, some LR recovery strategies require making changes to the state stack. Translating these changes into equivalent actions in the XLC(k) parser may be awkward (requiring modification of the machine's call stack) if the top-down component of the parser is coded as recursive functions. For this reason, we favour a recovery strategy that modifies the stream of unread tokens to correct the parse. Röhrich's method[10] based on error continuations, for example, would be very suitable and can easily be adapted for use with a XLC(1) parser.

6 Summary

The LAXLC(k) (1) construction algorithm has been used in the implementation of a recursive ascent-descent parser generator.
The parsing method is everything that has been claimed. Our parser generator accepts the full LALR(1) class of grammars and produces two sets of functions. The functions in the top-down component of the parser are human readable and manual insertion of semantic action code is easy. The programmer has the freedom to insert code at any free position in a rule's right hand side, while non-free positions are effectively inaccessible. The functions in the bottom-up component also have a simple structure and are simple to optimize.
Perhaps the best implementation of the parsing method would use a table-driven scheme for the bottom-up component and functions for the top-down approach. This would enable error recovery to be incorporated and would reduce the overall size of the parser.

References

[1] Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA (1985).

[2] Aretz, F.E.J.K. On a Recursive Ascent Parser. Information Processing Letters, 29, 3 (Nov. 1988), pp. 201-206.

[3] Barnard, D.T., and Cordy, J.R. SL Parses the LR Languages. Computer Languages 13, 2 (1988), pp. 65-74.

[4] Chapman, N.P. *LR Parsing: Theory and Practice.* Cambridge University Press, Cambridge, U.K. (1987).

[5] Demers, A.J. Generalized Left Corner Parsing. Proc. 4th Symposium on Principles of Programming Languages (1977), pp. 170-182.

[6] Fischer, C.N., and LeBlanc Jr., R.J. *Crafting A Compiler.* Benjamin/Cummings, Menlo Park, CA (1988).

[7] Purdom, P., and Brown, C.A. Semantic Routines and LR(k) Parsers. Acta Informatica, 14 (1980), pp. 299-315.

[8] Roberts, G.H. Recursive Ascent: An LR Analog to Recursive Descent. ACM SIGPLAN Notices, 23, 8 (Aug. 1988), pp. 23-29.

[9] Roberts, G.H. Another Note on Recursive Ascent. To appear in Information Processing Letters (1989).

[10] Röhrich, J. Methods for the Automatic Construction of Error Correcting Parsers. Acta Informatica, 13,2 (1980), pp. 115-139.