Recursive Ascent:
An LR Analog to Recursive Descent

George H. Roberts
13134 S. 125 E. Ave
Broken Arrow, OK 74011

Introduction

The usual implementation of a parser for an LL grammar is a recursive descent parser, a set of mutually recursive procedures, one for each production rule. This paper presents a similar implementation for the LR grammars: the recursive ascent parser. A recursive ascent parser is a set of mutually recursive procedures, one for each LR state (set of items). Figures 1 and 2 are cartoons of these implementations.

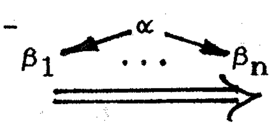
The basic operation of an LR parser is to repeatedly search for and execute the *action* associated with the current state of the parser and the current input symbol. Recursive ascent differs from other LR parser implementations in two important aspects. First, following a reduction, a recursive ascent parser searches a restricted set of the nonterminals. Second, its SHIFT and REDUCE *action*s are implemented by a subroutine linkage. These differences result in a structured, readable form for an LR parser.

The following shows why the search can be restricted, how the search is restricted, and the results of the restriction. The appendix contains the parser model and an example.

Figure 1. LL Grammar Implementation

Production Rule - $\alpha \rightarrow \beta_1 \ldots \beta_n$

Parse Tree Generation -



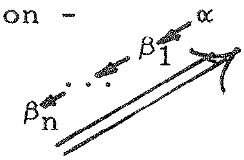
Recursive Descent Subroutine -

```
     recognize_α: call recognize_β1
                       ...
                  call recognize_βn
                  return
```

Figure 2. LR Grammar Implementation

State -
$$\begin{array}{lll} \alpha & \rightarrow & \ldots \cdot\beta_1 \ldots \\ \beta_1 & \rightarrow & \cdot\beta_2 \ldots \\ & \ldots & \\ \beta_{n-1} & \rightarrow & \cdot\beta_n \ldots \end{array}$$

Parse Tree Generation -



Recursive Ascent Subroutine -

closure_α: call closure_β_n

...

call closure_β_1

Why

The basic operation of an LR parser is to repeatedly search for and execute the *action* associated with the current state of the parser and the current input symbol. Traditionally, the search is conducted as a sparse 2-dimensional table search [JOH], but [PEN] showed that binary, sequential, and 1-dimensional searches are sufficient and faster.

Two distinct types of searches are conducted in LR parsing. The first is a search of the terminal symbols following an initial "shift" entry into a state. The second is a search of the nonterminal symbols following a "reduce" reentry . Only the nonterminal search is of interest here. While this search traditionally has been through all nonterminals that could appear in a state, the domain of the search can be much smaller.

LR states are composed of items of the form

$$\beta_i \rightarrow \chi_i \cdot \beta_j \omega_i$$

After an LR state is exited by shifting, that is, stacking an input symbol, β, and transferring to another state, the original state is reentered only after a reduction by a rule corresponding to an item of the form

$$\beta_i \rightarrow \cdot\beta\omega_i$$

(Reductions corresponding to items with $|\chi_i| > 0$ bypass this state.) On reentry, the symbols in the set

$$\{\beta_i : \beta_i \rightarrow \cdot\beta\omega_i \text{ is an item in the state}\}$$

are the only ones that can determine the next *action*, and the *action* search can be confined to this set.

How

In LR parsers, control transfers from one state to another by SHIFT and REDUCE *actions*. SHIFT stacks a return address and branchs to a new state entry. REDUCE adjusts the return address stack and returns. Following a reduction the "address returned to" initiates a search. In prior parsers reductions initiated searchs of all nonterminals that might appear in that state. Shortening the searches requires that a more precise return address be stacked. A standard subroutine call provides the required accuracy.

The construction of the recursive ascent subroutines starts with the usual LR state construction. Each state is then translated into the pseudocode template of Figure 3. The pseudocode is optimized, that is, code is rearranged to minimize jumps, case statements are implemented as binary or sequential searches, and code sequences are replaced by more efficient equivalent sequences. The result is then compiled. The appendix presents a simple example.

Result

Recursive ascent subroutines reflect parse tree generation in much the same way that recursive descent subroutines do. As such, they should be as readable as recursive descent subroutines. Compared to other implementations, recursive ascent parsers, in general, require more space but less time for searching.

Bibliography

[AHO] AHO, A.V., Ullman, J.D.
Principles of Compiler Design
Addison-Wesley 1977

[JOH] Johnson, S.C.
"YACC - Yet Another Compiler Compiler"
Computing Science Technical Report 32
AT&T Bell Laboratories, Murray Hill, NJ, 1975

[PEN] Pennello, T.J.
Very Fast LR Parsing
Sigplan Notices, 21, no. 7 (July 1986), pp 145-151.

Figure 3. Recursive Ascent Template

--items are of the form: $\beta_i \to \chi_i \cdot \beta_j \omega_i$

```
closure_ζ:                    --subroutine entry
push(SYMBOL,pop(INPUT))       --finish shift to this state
                              --(no shift into initial state)
case top(INPUT) of            --search terminals for this state

         --βj ∈ TERMINALS
  ...
  βj:
         --SHIFT to next state on βj
     call closure_βj             --recursive call
     case top(INPUT) of          --constrained search following
                                 --   reduction & return
       ...
       βi:  goto label_βi
       ...
     endcase

         --or REDUCE by α → β
     adjust(SYMBOL,|β|)
     push(INPUT,α)          --result placed on INPUT
     adjust(CONTROL,|β|-1)
     return
  ...
  default:   --error handling or default reduction
endcase


         --βj ∈ NONTERMINALS
...
label_βj:
         --SHIFT to next state on βj
  call closure_βj            --recursive call
  case top(INPUT) of         --constrained search following
                             --   reduction & return
    ...
    βi:  goto label_βi
    ...
  endcase

         --or ACCEPT  (initial state only)
  accept
...
```

Parser Model

The recursive ascent routines manipulate 3 data structures, INPUT, SYMBOL, and CONTROL. INPUT is the usual input stream - treated as a stack of symbols; SYMBOL is the usual stack used to hold shifted symbols; CONTROL is the usual stack used to save return addresses to be used after reductions. The usual stack functions

top(Ω)	return the top value on the Ω stack
pop(Ω)	return and delete the top value on the Ω stack
push(Ω,ω)	push the value ω on the Ω stack
adjust(Ω,ω)	discard the top ω values from the Ω stack

are applicable to each of these stacks. The usual parsing *actions* are represented by sequences of instructions in the recursive ascent subroutines:

accept -	accept	
error -	error	
shift Ω -	push(SYMBOL,pop(INPUT)) call Ω	{deferred until entering Ω}
reduce $\Omega \rightarrow \omega$ -	adjust(SYMBOL,$\|\omega\|$) push(INPUT,Ω) adjust(CONTROL,$\|\omega\|$-1) return	{result of a reduction is placed on the input}

Nonterminal searches textually follow *calls* to other subroutines (SHIFTS). In the following example the set searched is indicated at the call site. Where no set is indicated, there is no return to that point because of a stack adjustment.

Grammar

(adapted from [AHO] pages 200-208)

```
E' → E
E  → E + T
E  → T
T  → T * F
T  → F
F  → ( E )
F  → id
```

LR States

```
I0:  E' → ·E
     E  → ·E + T
     E  → ·T
     T  → ·T * F
     T  → ·F
     F  → ·( E )
     F  → ·id

I1:  E' → E ·
     E  → E ·+ T

I2:  E  → T ·
     T  → T ·* F

I3:  T  → F ·

I4:  F  → ( ·E )
     E  → ·E + T
     E  → ·T
     T  → ·T * F
     T  → ·F
     F  → ·( E )
     F  → ·id

I5:  F  → id·

I6:  E  → E + ·T
     T  → ·T * F
     T  → ·F
     F  → ·( E )
     F  → ·id

I7:  T  → T * ·F
     F  → ·( E )
     F  → ·id

I8:  F  → ( E ·)
     E  → E ·+ T

I9:  E  → E + T·
     T  → T ·* F

I10: T  → T * F·

I11: F  → ( E )·
```

Recursive Ascent Parser

```
I0:  if top(INPUT) = "(" then
         call I4             --{F}
     elseif top(INPUT) = id then
       call I5               --{F}
     else
       error
0/1: call I3                 --{T}
0/2: do
       call I2               --{T,E}
0/3: while top(INPUT) = T
     do
       call I1               --{E,E'}
0/4: while top(INPUT) = E
0/5: accept
```

```
I1: push(SYMBOL,pop(INPUT))
      if top(INPUT) = "+" then
        call I6
      else
        --reduce E' → E
        pop(SYMBOL); push(INPUT,E')
      return
1/1:
```

```
I2:  push(SYMBOL,pop(INPUT))
     if top(INPUT) = "*" then
       call I7
     else
       --reduce E → T
       pop(SYMBOL);push(INPUT,E)
       return
2/1:


I3:  --reduce T → F
     pop(INPUT);push(INPUT,T)
     return


I4:  push(SYMBOL,pop(INPUT))
     if top(INPUT) = "(" then
       call I4              --{F}
     elseif top(INPUT) = id then
       call I5              --{F}
else
       error
4/1: call I3                --{T}
4/2: do
       call I2              --{T,E}
4/3: while top(INPUT) = T
     do
       call I8              --{E}
4/4: forever


I5:  --reduce F → id
     pop(INPUT);push(INPUT,F)
     return


I6:  push(SYMBOL,pop(INPUT))
     if top(INPUT) = "(" then
       call I4              --{F}
     elseif top(INPUT) = id then
       call I5              --{F}
     else
       error
6/1: call I3                --{T}
6/2: do
       call I9              --{E}
6/3: forever
```

```
I7:  push(SYMBOL,pop(INPUT))
     if top(INPUT) = "(" then
       call I4              --{F}
     elseif top(INPUT) = id then
       call I5              --{F}
     else
       error
7/1: call I10
7/2:


I8:  push(SYMBOL,pop(INPUT))
     if top(INPUT) = ")" then
       call I11
8/1: elseif top(INPUT) = "+" then
       call I6
     else
       error
8/2:


I9:  push(SYMBOL,pop(INPUT))
     if top(INPUT) = "*" then
       call I7
     else
       --reduce E → E + T
       adjust(SYMBOL,3)
       push(INPUT,E)
       adjust(CONTROL,2); return
9/1:


I10: --reduce T → T * F
     pop(INPUT)
     adjust(SYMBOL,2); push(INPUT,T)
     adjust(CONTROL,2); return


I11: --reduce F → ( E )
     pop(INPUT)
     adjust(SYMBOL,2); push(INPUT,F)
     adjust(CONTROL,2); return
```