

Recursive Descent Parsing

Recursive Descent is a technique for building “special purpose” LL(1) parsers.

LL(1) = Left to right parse,
building Left-most derivation,
with 1 symbol look-ahead.

- Parser is a set of mutually recursive procedures, with (roughly) one corresponding to each non-terminal.
- Each procedure decides which rule to use, then looks for the symbols in the rhs of that rule.
- Matches the rhs from left to right
⇒ Expand left-most non-terminal in parse tree.
- Uses properties of LL(1) grammars to decide which rule to use at each step.
- Must be able to choose rule by looking at next input.
- Call corresponding procedure to recognise non-terminals.
- Call scanner to recognise terminals (and remove white space, comments, etc.).

Parser procedures

- Goal of the procedure corresponding to the start symbol of the grammar, S , is to read a sequence of input symbols that form a string in the language $L(S)$, and to return a pointer to the root of the parse tree for this string.
- Production body can be thought of as a sequence of goals that must be fulfilled in order to find a string in $L(N)$, where N is the head of the production.

Parser procedures

Example:

- $B \rightarrow \epsilon$
- $B \rightarrow (B) B$

This says that one way to find a string of balanced parentheses is to fulfill the following four goals in order.

1. Find character '('
 2. Find a string of balanced parentheses
 3. Find the character ')'
 4. Finally, find another string of balanced parentheses
- Terminal goal is satisfied if we find that this terminal matches the next input symbol, can't be satisfied if the next input symbol is something else.
 - A non-terminal goal is satisfied by calling the procedure for that non-terminal.
 - Only hard part is to decide whether to satisfy a goal of finding a B by using production 1, which succeeds immediately, or by using production 2.

Structure of Parser Procedures

For each non-terminal, N , procedure ParseN tries to recognise an instance of N as a prefix of the remaining input.

If N is defined as $N \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$, then ParseN is:

```
PROCEDURE ParseN;  
BEGIN  
    IF (Input may contain an  $\alpha_1$ ) THEN  
         $parse(\alpha_{1,1}); \dots; parse(\alpha_{1,|\alpha_1|})$   
    ELSIF (Input may contain an  $\alpha_2$ ) THEN  
        ...  
    ELSIF (Input may contain an  $\alpha_n$ ) THEN  
         $parse(\alpha_{n,1}); \dots; parse(\alpha_{n,|\alpha_n|})$   
    ELSE Error END IF  
END ParseN
```

Where $parse(x)$ is $Parse\ x$ if x is a non-terminal
 $Scan(x)$ if x is a terminal

To test whether “Input may contain an α_i ”? (I.e. “Can an instance of α_i be a prefix of the remaining input?”.)

- If $\alpha_i \not\Rightarrow^* \epsilon$: $next \in first(\alpha_i)$.
- If $\alpha_i \Rightarrow^* \epsilon$: TRUE. Must be the last rhs tested.

Building a Parser for Nested Lists

- LL(1) grammar for nested lists:

$$\begin{array}{lll}
 List \rightarrow & "(" RestList & (1) \\
 RestList \rightarrow & ")" \mid ListBody "(" & (2), (3) \\
 ListBody \rightarrow & ListElt RestBody & (4) \\
 RestBody \rightarrow & \epsilon \mid ", " ListBody & (5), (6) \\
 ListElt \rightarrow & number \mid List & (7), (8)
 \end{array}$$

- Satisfies the two requirements to ensure that we can choose rule by looking at next input.
- Parser will have one procedure for each nonterminal: *ParseList*, *ParseRestList*, etc.
- Input is a sequence of *symbols*, *ss*.
- Symbols are recognised by a *scanner*, which also removes white space.

The symbols used are:

<u>Symbol Kind</u>	<u>Symbol</u>
lparsym	"("
rparsym	")"
commasym	", "
numbersym	(0 1 2 3 4 5 6 7 8 9) ⁺

- Use Start, Current, Advance, Scan and Error on symbol stream.

Parser for Nested Lists: List

Rule: $List \rightarrow "(" RestList$ (1)

```
PROCEDURE ParseList;  
BEGIN  
    IF Current(ss) = lparensym THEN  
        Advance(ss);  
        ParseRestList  
    ELSE Error END IF  
END ParseList;
```

Rule: $RestList \rightarrow ")"$ | $ListBody ")"$ (2), (3)

```
PROCEDURE ParseRestList;  
BEGIN  
    IF Current(ss) = rparensym THEN  
        Advance(ss)  
    ELSE  
        ParseListBody;  
        Scan(rparensym)  
    END IF  
END ParseList;
```

Parser for Nested Lists: ListBody

Rule: $ListBody \rightarrow ListElt RestBody$ (4)

```
PROCEDURE ParseListBody;  
BEGIN  
    ParseListElt;  
    ParseRestBody  
END ParseListBody;
```

Rule: $RestBody \rightarrow \epsilon \mid \text{“,” } ListBody$ (5), (6)

```
PROCEDURE ParseRestBody;  
BEGIN  
    IF Current(ss) = commasym THEN  
        Advance(ss);  
        ParseListBody  
    ELSE  
        SKIP  
    END IF  
END ParseRestBody;
```

Parser for Nested Lists: ListElt

Rule: $ListElt \rightarrow number \mid List$ (7), (8)

```
PROCEDURE ParseListElt;  
BEGIN  
    IF Current(ss) = numbersym THEN  
        Advance(ss);  
    ELSE  
        ParseList  
    END IF  
END ParseListElt;
```

Calling the parser:

```
Start(ss);  
ParseList;  
Scan(eofsym)
```

This just accepts or rejects the input.

What happens if parse fails?

Building a Parse Tree

If input is accepted, parse tree is implicit in procedure calling pattern — which is lost by the time parsing is finished.

How can we construct the parse tree?

- Design data type to represent parse trees.
Assume $\text{TREE}(N, t_1, \dots, t_n)$ builds a tree with root N and subtrees t_1, \dots, t_n (where $n \geq 1$).
(And `ErrorTree` is a distinguished tree value indicating an error.)
- Add calls in the parser to construct the parse tree.
- Can construct the parse tree either:
 - (1) As the parser works down the tree.
 - (2) As the parser exits the called procedures.

Prefer (2): clean interface to tree type, easy to handle failure.

- Insert code to collect the components corresponding to the RHS of the rule applied.
- Add code to “apply rule” at end of code that checks each rule.
- Builds tree bottom up, each time we exit a procedure , return a subtree. Gather these up at end of next level up.
- Scanner returns symbol value as well as symbol kind.
- Scan procedure returns symbol recognised.
- Parser procedure returns tree for string it recognised;
return empty/error tree if parse fails