Backtracking (1)

- Consider the Grammar:

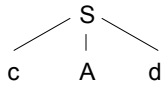
```
S → cAd
A → ab | a
```

- Given the input string **cad**, we try and construct the parse tree. We initially start by creating the Root of the tree from S (the current token is **c** in the input string:

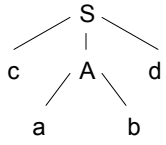
S

Backtracking (2)

- Clearly the current token **c** does not match **S** so we expand **S** using the first (and only rule):

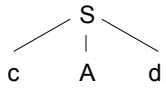


- The leftmost leaf matches our input symbol **c** so we proceed to the next one **a** and consider the next leaf **A**. In expanding **A**, we have two alternatives. Having no preference, we arbitrarily choose the first one to get the tree:



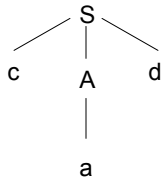
Backtracking (3)

- We have a match for the current token **a**, so we proceed with the next one **d**. Looking at the next leaf **b**, we see that the token does not match, so we must have expanded using the wrong production. We must backtrack to the state before the production was chosen – the current symbol is set back to **a**, and the tree:



Backtracking (4)

- We now try the other alternative and expand the tree to:



- The current input **a** matches the leaf. We move to the next token **d** and the next leaf, which match too. All the input has been consumed and we have completed the parse successfully.

Notes (1)

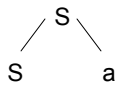
- The parsing methods we have seen are called **recursive-descent** parsers.
- Grammars can be rearranged to eliminate the need for backtracking. Parsers for such grammars are called **predictive parsers**.
- A left-recursive grammar (productions of the type A → Aα) can cause a recursive-descent parser to go into infinite loops, even if it has backtracking.
- Consider the grammar:

  S → Sa
  s → ε

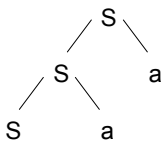
- Our task is to parse the string **aaaaaa**.

Notes (2)

- The current token is **a**. We expand the **S** node to get the tree:

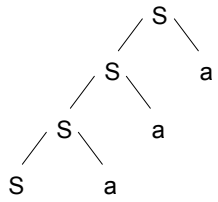


- The first leaf is a non-terminal so we expand again. We have two choices so we arbitrarily choose the first:



Notes (3)

- Again, the first leaf is a non-terminal so we expand again. We have two choices so we arbitrarily choose the first:



- The problem is that the tree will continue growing indefinitely without ever consuming any input (the terminating condition is never achieved).

Eliminating Left-Recursion

- A grammar is left-recursive if it has a derivation of the type $A \Rightarrow^{+} A\alpha$.
- As we have seen, top-down parsers cannot handle left-recursion, so we need a transformation these grammars into right recursive ones.
- A left-recursive production of the form

$$A \rightarrow A\alpha \mid \beta$$

- Can be rewritten as:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Example

- Consider the following grammar for arithmetic expressions

```
E → E + T | T
T → T * F | F
F → ( E ) | Ident
```

- The grammar is rewritten as:

```
E  → TE'
E' → +TE' | ε
T  → FT'
T' → *FT' | ε
F  → ( E ) | Ident
```


Non-Immediate Left-Recursion

- Immediate left-recursion involves productions that involve left-recursive derivations in one step:

$$A \rightarrow A\alpha$$

- There are cases where left-recursion may occur after more than one derivational steps. For example, the following grammar is not immediately left recursive:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

- The Non-Terminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$

Multiple A-Productions

- In the previous example, we considered eliminating a single instance of left-recursions from an A-Production ($\mathbf{A} \rightarrow \mathbf{A}\alpha \mid \beta$)
- No matter how many left-recursive A-productions there are ($\mathbf{A} \rightarrow \mathbf{A}\alpha_1 \mid \mathbf{A}\alpha_2 \mid \ldots \mid \mathbf{A}\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$), all we have to do is replace the productions by:

$$\mathbf{A} \rightarrow \beta_1\mathbf{A}' \mid \beta_2\mathbf{A}' \mid \ldots \mid \beta_n\mathbf{A}'$$
$$\mathbf{A}' \rightarrow \alpha_1\mathbf{A}' \mid \alpha_2\mathbf{A}' \mid \ldots \mid \alpha_m\mathbf{A}' \mid \varepsilon$$

Algorithm for Eliminating Left-Recursion (1)

- This algorithm is guaranteed to eliminate left-recursion for grammars having:
  - No Cycles: $A \Rightarrow^+ A$
  - No Empty Productions: $A \rightarrow \varepsilon$
- The input of the Algorithm is a grammar G with No Cycles or Empty Productions.
- The output of the Algorithm is the equivalent grammar without left-recursion.

Algorithm for Eliminating Left-Recursion (2)

Arrange the Non-Terminals in some order A_1, A_2, … , A_n.

```
For i  = 1 to n
   For j = 1 to (i-1)
         replace every production of the form
               Ai → Ajγ by the productions
               Ai → δ1γ | δ2γ | … | δkγ, where
               Aj → δ1 | δ2 | … | δk
   Next j
   Eliminate the immediate left-recursion
         among the Ai productions
Next i
```


Example on a Grammar (1)

- Consider the grammar:

  S → Aa | b
  A → Ac | Sd | ε

- Note: technically, the algorithm is not guaranteed to work because of the A → ε empty production, though in this case it does.
- We order the non-terminals as **S, A**.
- Starting from **S (i = 1)**, the inner **j** loop does not execute and we have to eliminate immediate left recursion. But there is no immediate left recursion among **S** productions so nothing happens.

Example on a Grammar (2)

- For the step **`i = 2`**, the inner **`j`** loop, substitutes all occurrences of **`S`** in the A-Productions, producing the following:

```
A → Ac | Aad | bd | ε
```

- In this case we do have left recursion so proceed to remove it, producing:

```
S  → Aa | B
A  → bdA′ | A′
A′ → cA′ | adA′ | ε
```


Left-Factoring (1)

- Left-factoring is a grammar transformation that produces grammars suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a non-terminal **A**, we defer the decision until we have seen enough input to make the right choice.
- Consider the two productions:

```
stmt → "if" expr "then" stmt "else" stmt
stmt → "if" expr "then" stmt
```

- On seeing the input "IF", it is not clear which production to use to expand the statement.

Left-Factoring (2)

- In general, given two productions:

  $A \rightarrow \alpha\beta_1$
  $A \rightarrow \alpha\beta_2$

- We may rewrite the rules as:

  $A \rightarrow \alpha A'$
  $A' \rightarrow \beta_1 \mid \beta_2$

- This way we have only one choice for expanding **A**.

Algorithm for Left-Factoring

Input: *Grammar G.*
Output: *An equivalent left-factored grammar.*
Method:

1. For each non-terminal A, find the longest prefix α common to all the alternatives (2 or more).
2. If $\alpha \neq \varepsilon$ (there IS a common prefix)
3. Replace all the productions for A having the prefix α ($A \rightarrow \alpha\beta_1$, $A \rightarrow \alpha\beta_2$, … , $A \rightarrow \alpha\beta_n$) with:

   $A \rightarrow \alpha A'$
   $A' \rightarrow \beta_1 , \beta_2 , \ldots , \beta_n$

Operator Precedence Parsing of Expressions

- The basis of Operator Precedence Parsing is assigning a priority to each operator in an expression. For example:

```
3 * 4 + 5 = (3 * 4) + 5
```

- Parenthesis may be used to change and visually emphasis precedence:

```
3 * (4 + 5) ≠ 3 * 4 + 5
```


Associativity of Operators

- Associativity for an operator, say ⊗, may be left-associative or right associative:
  - Left-Associative: expression `x ⊗ y ⊗ z` is evaluated as `(x ⊗ y) ⊗ z`
  - Right-Associative: expression `x ⊗ y ⊗ z` is evaluated as `x ⊗ (y ⊗ z)`
- For example, given: `a = 4, b = 5, c = 2`, the expression `a – b – c` may be evaluated as:
  - `(a – b) – c = (4 – 5) – 2 = -3`, or
  - `a – (b – c) = 4 – (5 – 2) = 1`

Operator Precedence for Parsing Simple Expressions (1)

- Given the following grammar for simple expressions:

```
expr        ::=  <primary> {<op> <expr>};
primary     ::=  <ident>
               | <constant>
               | '-' <primary>
               | '+' <primary>
               | '(' <expr> ')';
```


Parsing the Primaries (1)

```
Function Parse_Primary(token)
  if token = IDENTIFIER then
      // create a leaf with the identifier
      node = IdentLeaf(token)
  else if token = CONSTANT then
      // create a leaf with the constant
      node = ConstantLeaf(token)
  else if token = ADD_OP or token = SUB_OP then
      remember the operator
      lookahead = GetNextToken()
      // parse the primary following the unary op
      primary_node = Parse_Primary(lookahead)
      // combine the unary op and primary in a node
      node = UnaryOpNode(saved operator, primary_node)
```

Parsing the Primaries (2)

```
else if token = OPEN_BRACKET then
     lookahead = GetNextToken()
     // the Parse_Expression function returns the expression
     // node and consumes the next lookahead character
     // which should be the closing bracket
     node = Parse_Expression(lookahead)

     if lookahead <> CLOSE_BEACKET then
            Error – Missing Closing Bracket
     else // read the lookahead
            lookahead = GetNextToken()
     end if
else
     // an invalid character
     Error – Invalid character in primary
end if
```


Parsing the Expression (1)

```
Function Parse_Expression(token, LeftPriority)
   lhs = Parse_Primary(token)
   if Primary Parsed Correctly then
        EndOfExpression = FALSE

        While       (token is an oparator)
               AND (NOT EndOfExpression)

               // see if this operator has a higher priority
               // than the one to its left. The priority
               // of the token to the left is passed as an
               // argument to this function
               if    (Priority of current op)
                   > (LeftPriority) then
                       // this op takes precedence – parse the rhs
                       Remember this operator
                       token = GetNextToken()
```

Parsing the Expression (2)

```
            rhs = Parse_Expression(token,
                                   Priority of current op)

            // combine the lhs to the rhs to act as an
            // lhs for further operators to the right
            lhs = BinaryOpNode(lhs, Current Op, rhs)
      else
            // left op has higher or equal priority
            EndOfExpression = TRUE
      end if
   end while

   // make expression node in parse tree
   node = ExpressionNode(lhs)

   return the lookahead at expression node
```


Example (1)

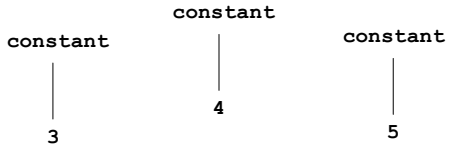
- Consider the parsing of the expression **"3 + 4 * 5 / 6"**
- The first call to Parse_Expression would be **ParseExpression(3, 0)**
  - Since there are no operators to the left we pass 0 (the lowest possible priority).
  - The call to **Parse_Primary** will create a constant leaf containing **3**.
  - The operator read is a **"+"** which has a higher priority than the one to the left (which actually doesn't exist)
  - An re-invocation to **Parse_Expression** is made to parse the rhs.

Example (2)

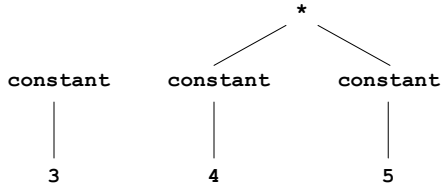
- The call to parse the rhs would be Parse_Expression (4, Priority of “+”).
  - **`Parse_Primary`** will create a constant leaf for **`"4".`**
  - The operator here is **`"*"`** which has a higher priority than the one on it’s left **`"+"`**
  - Another call to **`Parse_Expression`** is made to parse the rhs.
- The call to parse the rhs would be Parse_Expression(5, Priority of “*”).
  - **`Parse_Primary`** will create a constant leaf for **`"5".`**
  - The operator here is **`"/"`** which does not have a priority greater than the one to the left **`"*"`** so **`EndOfExpression`** is set to true, the function terminates and control passes to the previous call of **`Parse_Expression(4, Priority of "+")`**

Example (3)

- So far the tree is:

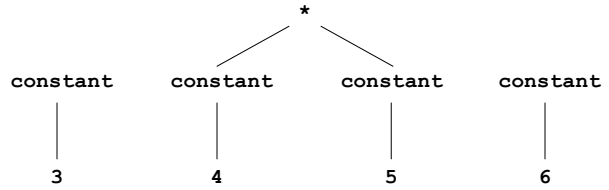


- Now **`Parse_Expression`** has both the LHS and RHS of the multiplication and can join the nodes to get:



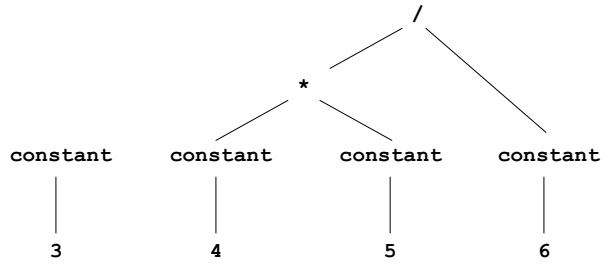
Example (4)

- The multiplication becomes the lhs for the current token "/". This is greater then the priority of the current left operator "+" so a new call to parse is made:
- Parse_Expression(6, Priority of "/")
  - The function will return after creating the constant leaf for "6".



Example (5)

- The function will now return to the previous call that will join the lhs and rhs using the division node:



Example (6)

- Again the function will return to the previous call that will join the lhs and rhs using the addition node: