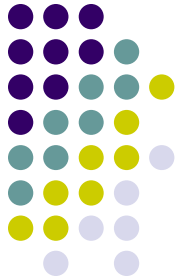
Recursive Descent Parsing

- Top-down parsing: build tree from root symbol
- Each production corresponds to one recursive procedure
- Each procedure recognizes an instance of a non-terminal, returns tree fragment for the non-terminal

General model

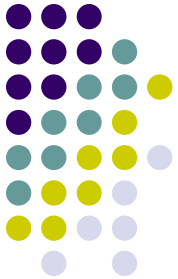


- Each right-hand side of a production provides body for a function
- Each non-terminal on the rhs is translated into a call to the function that recognizes that non-terminal
- Each terminal in the rhs is translated into a call to the lexical scanner. Error if the resulting token is not the expected terminal.
- Each recognizing function returns a tree fragment.

Example: parsing a declaration

- FULL_TYPE_DECLARATION ::=
- type DEFINING_IDENTIFIER is TYPE_DEFINITION;
- Translates into:
  - get token type
  - Find a defining_identifier -- function call
  - get token is
  - Recognize a type_definition -- function call
  - get token semicolon
- In practice, we already know that the first token is type, that's why this routine was called in the first place! Predictive parsing is guided by the next token

Example: parsing a loop



- FOR_STATEMENT ::=
  ITERATION_SCHEME loop STATEMENTS end loop;

```
Node1 := find_iteration_scheme;          -- call function
get token loop
List1 := Sequence of statements          -- call function
get token end
get token loop
get token semicolon;
Result := build loop_node with Node1 and List1
return Result
```

Complications

- If there are multiple productions for a non-terminal, we need a mechanism to determine which production to use

  IF_STAT ::= if COND then Stats end if;

  IF_STAT ::= if COND then Stats ELSIF_PART end if;

When next token is if, can't tell which production to use.

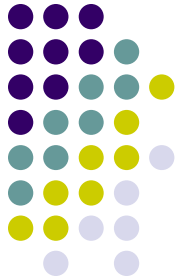
Solution: factorize grammar

- If several productions have the same prefix, rewrite as single production:

- IF_STAT ::= if COND then STATS [ELSIF_PART] end if;

- Problem now reduces to recognizing whether an optional
- Component (ELSIF_PART) is present

Complication: recursion



- Grammar cannot be left-recursive:
- E ::= E + T | T
- Problem: to find an E, start by finding an E…
- Original scheme leads to infinite loop: grammar is inappropriate for recursive-descent

Solution: remove left-recursion

- E ::= E + T | T means that eventually E expands into

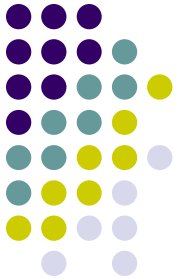
  T + T + T ….

- Rewrite as:
  - E ::= TE’
  - E’ ::= + TE’ | *epsilon*
- Informally: E’ is a possibly empty sequence of terms separated by an operator

Recursion can involve multiple productions

- A ::= B C | D
- B ::= A E | F
- Can be rewritten as:
- A ::= A E C | F C | D
- And then apply previous method
- General algorithm to detect and remove left-recursion from grammar (see ASU)

Further complication



- Transformation does not preserve associativity:
- E ::= E + T | T
- Parses a + b + c as (a + b) + c
- E ::= TE', E' ::= + TE' | ***epsilon***
- Parses a + b +c as a + (b + c)

- Incorrect for a - b – c : must rewrite tree

In practice: use loop to find sequence of terms

```
Node1 := P_Term;    --  call function that recognizes a term
loop
        exit when Token not in Token_Class_Binary_Addop;
        Node2 := New_Node (P_Binary_Adding_Operator);
        Scan;                                          --  past operator
        Set_Left_Opnd    (Node2, Node1);
        Set_Right_Opnd (Node2, P_Term);   --  find next term
        Set_Op_Name (Node2);
       Node1 := Node2;              --  operand for next operation
end loop;
```