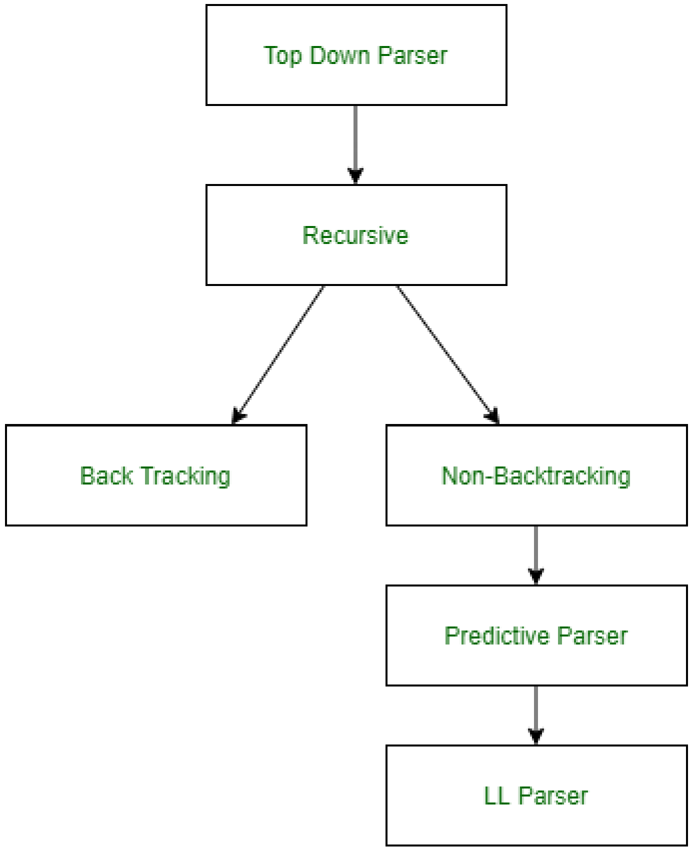
CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

A short classification



Parsing

- Predictive Parsing
  - Remember we still have three productions?

```
type → simple | ↑ id | array [ simple ] of type
```

  - For each production P, we wish to construct the set FIRST(P), consisting of those tokens (i.e., terminals) that can appear as the first symbol of a string derived from the RHS of P.
  - See the table (next page) for an example

Parsing

Production	FIRST
type → simple	{ integer, char, num }
type → ↑ id	{ ↑ }
type → array [simple] of type	{ array }
simple → integer	{ integer }
simple → char	{ char }
simple → num dotdot num	{ num }

[Part2]

- Three productions with *type* as LHS have disjoint FIRST sets
- Three productions with *simple* as LHS have disjoint FIRST sets
- Thus, predictive parsing can be used.
- We process the input left to right and call the current token **lookahead** since it is **how far we are looking ahead** in the **input** to determine the production to use.

[Part1]

- So, FIRST is actually defined on strings not productions
- When I write FIRST(P), I really mean FIRST(RHS).
- Let α be a string of terminals and/or non-terminals.
- FIRST(α), is the set of terminals that can appear as the first symbol in a string of terminals derived from α.
- In other words, if α is ε or α can derive ε, then ε is in FIRST(α)
- So, given α, we find all strings of terminals that can be derived from α and pick off the **first** terminal from each string to build up our FIRST(α)
- **Question:** How do we calculate FIRST(α)?
- **Answer:** Wait until chapter 4 for a formal algorithm. For these simple examples it is reasonably clear.

Parsing

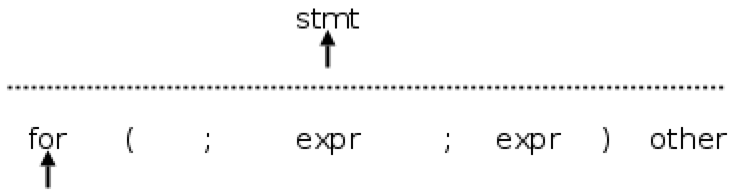
- When to Use ε-productions
  - Not all grammars are as friendly as the last example.
  - The first complication is that, when ε occurs as a RHS
    - If this happens or,
    - if the RHS can generate ε,
    - then ε is (should be) included in FIRST.
  - The rule is that:
    - if **lookahead** symbol is not in FIRST(expr) of any production with the desired LHS,
    - we should use the (unique!) **production** (with **that LHS**) that **has ε** in **FIRST(**expr).
    - That means the “epsilon production” is used
    - See the previous First() table, we do not have the “epsilon production”

Parsing

- When to Use ε-productions (Cont.)
  - There is a C language like example, productions are: (Fig. 2.16 in the book)

```
stmt → expr ;
     | if ( expr ) stmt
     | for ( optexpr ; optexpr ; optexpr ) stmt
     | other
optexpr → expr | ε
```

  - Here is the beginning of the movie to build the final parsing tree (not the full movie)
    - Up: Tree we are going to generate
    - Down: Input string
    - Step1:



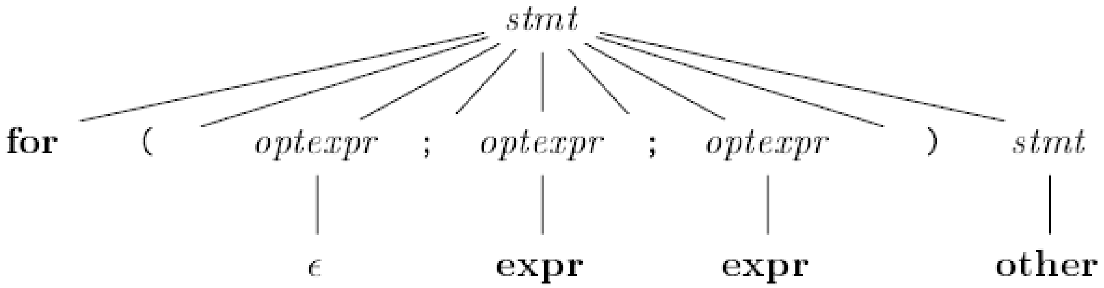


Figure 2.17: A parse tree according to the grammar in Fig. 2.16

Parsing

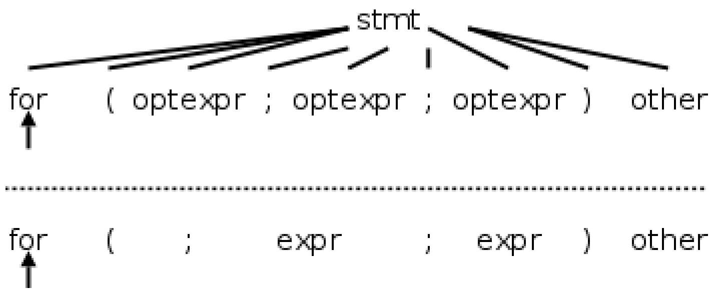
- Based on this example:

  for (; expr ; expr) other

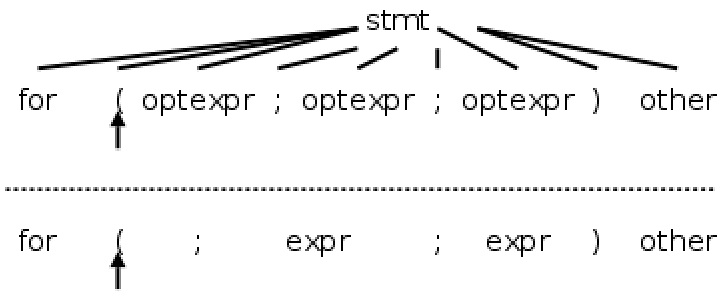
- Initially, the terminal for is the lookahead symbol and the known part of the parse tree consists of the root, labeled with the starting nonterminal “stmt”
- The objective is to construct the remainder of the parse tree in such a way that the string generated by the parse tree matches the input string.
- For a match to occur, the nonterminal stmt, must derive a string that **starts** with the lookahead symbol **for**. (because we need to match)
- There is just one production for stmt that can derive such a string (for), so we select it, and construct the children of the root labeled with the symbols in the production body.
- We go to Step 2

Parsing

- Step2:



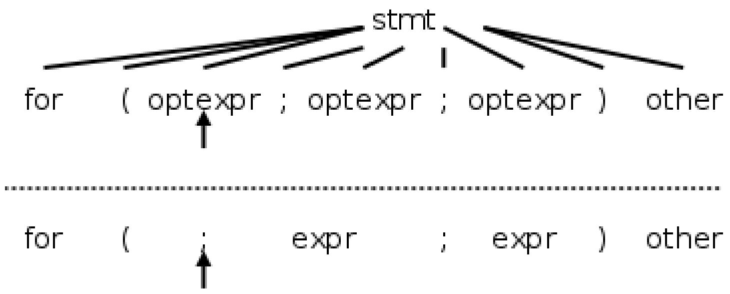
- Step3:



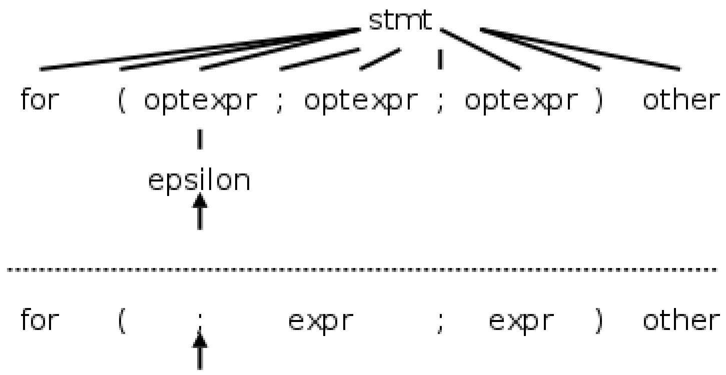
- The next terminal in the input becomes the new lookahead symbol, and the next child in the parse tree is considered
- Then, “(” is a terminal, in step 3

Parsing

- Step4



- Step5: (Note: I choose epsilon!)



- A further advance will take the arrow in the parse tree to the child labeled with nonterminal optexpr and take the arrow in the input to the terminal ";"
- At the nonterminal node labeled optexpr, we repeat the process of selecting a production for a nonterminal.
- With nonterminal optexpr and lookahead ";", the epsilon-production is used, since ";" does not match…
- The only other production for optexpr, which has terminal expr as its body.

The full story will be revealed in chapter 4

Parsing

- Designing a Predictive Parser
  - A predictive parser is a recursive descent parser with no backtracking or backup. It is a top-down parser that does not require backtracking.
  - At each step, the choice of the rule to be expanded is made upon the next terminal symbol.

Parsing

- Designing a Predictive Parser (Cont.)
  - They are recursive descent parsers we go top-down with **one procedure for each nonterminal**.
  - Do remember that to use predictive parsing, **we must have disjoint FIRST sets** for all the productions having a given nonterminal as LHS.
  - Recall that a predictive parser is a program consisting of a procedure for every nonterminal.
  - The procedure for nonterminal A does two things.
    - It decides which A-production to use by examining the lookahead symbol. The production with body alpha, (where α is not epsilon) is used if the lookahead symbol is in FIRST(α).

Parsing

- Designing a Predictive Parser (Cont.)
  - The procedure for nonterminal A does two things. (Cont.)
    - The procedure then mimics the body of the chosen production.
      - That is, the symbols of the body are “executed" in turn, from the left.
      - A nonterminal is executed" by a call to the procedure for that nonterminal
      - A terminal **matching** the lookahead symbol is “executed“ by reading the next input symbol.

Parsing

- Left Recursion
  - For the first production the RHS begins with the LHS. This is called left recursion.
  - If a recursive descent parser would pick this production, the result would be that the next node to consider is again expr and the lookahead has not changed.
  - An infinite loop occurs.
    - expr→ expr + term → expr + term +term → …
  - Also note that the first sets are not disjoint

```
expr → expr + term
expr → term
```

Parsing

- Note that this is **NOT** a problem with the grammar, but is a limitation of predictive parsing
- For example if we had the additional production
    - term → x
- Then, it is easy to construct the unique parse tree for:
    - x + x
- We still cannot use predictive parsing

expr → expr + term
expr → term
term → x

Parsing

- If the grammar were instead:

Set1

```
expr → term + expr
expr → term
```

- It would be right recursive, which is not a problem. But the first sets are still NOT disjoint and it would become right associative.

Consider, instead of the original (left-recursive) grammar, the following replacement

Set2

```
expr → term rest
rest → + term rest
rest → ε
```

Both sets of productions generate the same possible token strings, namely

```
term + term + ... + term
```

Parsing

- In Set2, it is, the same, called right recursive since the RHS ends (has on the right) the LHS
- If you draw the parse trees generated, you will see that, for left recursive productions, the tree grows to the left; whereas, for right recursive, it grows to the right.
- In general, for any nonterminal A, and any strings α, and β (α and β cannot start with A), we can replace the pair of productions in this way!

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

with the triple

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

- Where R is a nonterminal not equal to A and not appearing in α or β, i.e., R is a new nonterminal. (newly introduced)

Parsing

- For the example above:
  - A is expr,
  - **R is rest,**
  - α is + term, and
  - β is term.

New Form

$A \to \beta\ R$

$R \to \alpha\ R \mid \varepsilon$

Original

expr → term rest

rest → + term rest

rest → ε

A Translator for Simple Expressions

- Ch.2.5 TBD