CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

- Writing a Grammar (4.3)
  - ~~Lexical Versus Syntactic Analysis (4.3.1)~~
  - ~~Eliminating Ambiguity (4.3.2)~~
  - ~~Elimination of Left Recursion (4.3.3)~~
  - ~~Left Factoring (4.3.4)~~
- Top-Down Parsing (4.4)
  - **Recursive Decent Parsing (4.4.1)**
- Bottom-up Parsing (4.5)
- Finally…

Recursive Decent Parsing (4.4.1)

- A recursive-descent parsing consists of a set of procedures, one for each nonterminal.
- Yes, we are doing selections based on the productions on “nonterminal(s)”
- Execution begins with the procedure for the start symbol (root), which **halts** and announces **success** if its procedure body scans the **entire input string**
- Remember the A-production we learned earlier? The procedure is non-deterministic and it begins by choosing the A production. But how?

Recursive Decent Parsing (4.4.1)

- There is one more thing.
- The general recursive-descent may require backtracking; that is, it may require repeated scans over the input.
- However, backtracking is rarely needed to parse programming language constructs and that is why such kind of parsers are not frequently seen.
- Here is a general (very generic) guideline of the top-down parser

(See the next page for detail)

Recursive Decent Parsing (4.4.1)

- A “generic” procedure in describing how to process a single non-terminal A and its production

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

Recursive Decent Parsing (4.4.1)

- To allow backtracking, the code in the previous page has to be modified a little bit!
  - Firstly, we cannot choose a unique A-production at line (1), so we must try each of several productions in some “order”.
  - Second, failure at line (7) is not the ultimate failure, but it suggests us that we need to return to line (1) and try another A-production.
    - We declare that an input error has been found, only if there are no more A-productions to we can try
  - We are going to run an example in the next page
  - Still remember we have a “movie”, 2 panes, in the chapter 2? The upper part is the tree building process, and the lower part is the scanning of the input string?

Recursive Decent Parsing (4.4.1)

- Example1(from the book):

Example 4.29 : Consider the grammar

$$\begin{array}{lcl} S & \rightarrow & c\ A\ d \\ A & \rightarrow & a\ b \mid a \end{array}$$

- To construct a parse tree top-down for the input string ***w = cad***,
- It begins with a tree, consisting of a single node labeled S, and the input pointer pointing to ***c***, the ***first symbol*** of ***w***.
- S has **only one production**, so we use it directly to expand S and obtain the tree of Fig. 4.14(a).
- (see the next page)

Recursive Decent Parsing (4.4.1)

- The leftmost leaf, labeled ***c***, matches the **first** symbol of input **w**
- So we advance the "input pointer (not the tree nodes!)" to **a**, (w=cad)
- Since we previously use the production (and is the only production),

for S, S→cAd, now, we will expand A, (because we got a match to **c** already) using the first alternative A → a b to obtain the tree of Fig. 4.14(b).

Example 4.29 : Consider the grammar

$$S \rightarrow c\ A\ d$$
$$A \rightarrow a\ b \mid a$$

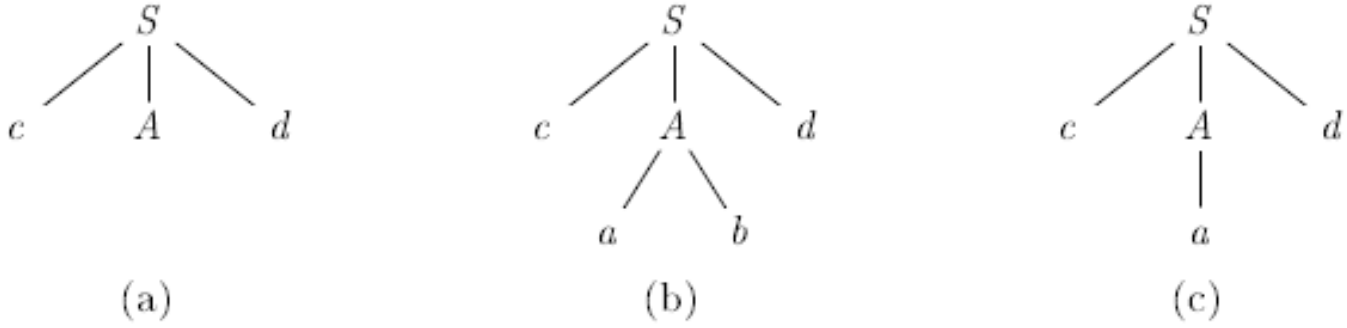

Figure 4.14: Steps in a top-down parse

Recursive Decent Parsing (4.4.1)

- We have a match for the second input symbol, a, so we advance the input pointer to d, the third input symbol in "***cad***"
- We compare d against the next leaf, labeled ***b***. It is NOT a match!
- We report failure and go back to A to see whether there is another alternative production for A that has not been tried
- In going back to A, we must reset the "input pointer" to position 2, "***cad***"

the position it had when we first came to A

- Now we choose a different production for A, we need to be able

to reset the input pointer to where it was when we first reached line (1).

$$S \rightarrow c\ A\ d$$
$$A \rightarrow a\ b \mid a$$

Recursive Decent Parsing (4.4.1)

- Now we choose A→a, The leaf a matches the second symbol of w and the leaf d matches the third symbol.
- Since we have produced a parse tree for w, we halt and announce successful **completion** of parsing.

Bottom up parsing

- A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves, and working up towards the root
- For bottom up parsing, we are not fearful of left recursion as we were with top down. (Because we already know everything, the leaves in the tree). The left recursion for top-down parsing, sometimes, it might be “endless”, if you are in a bad luck
- Our first example will use the left recursive expression grammar to introduce this
- (See the next page for example)

Bottom up parsing

- Considering there is a grammar (or a set of productions)

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

- If, the leaves is id*id, we can build up the parsing tree in the ***reversed*** order. We look for the non-terminals!

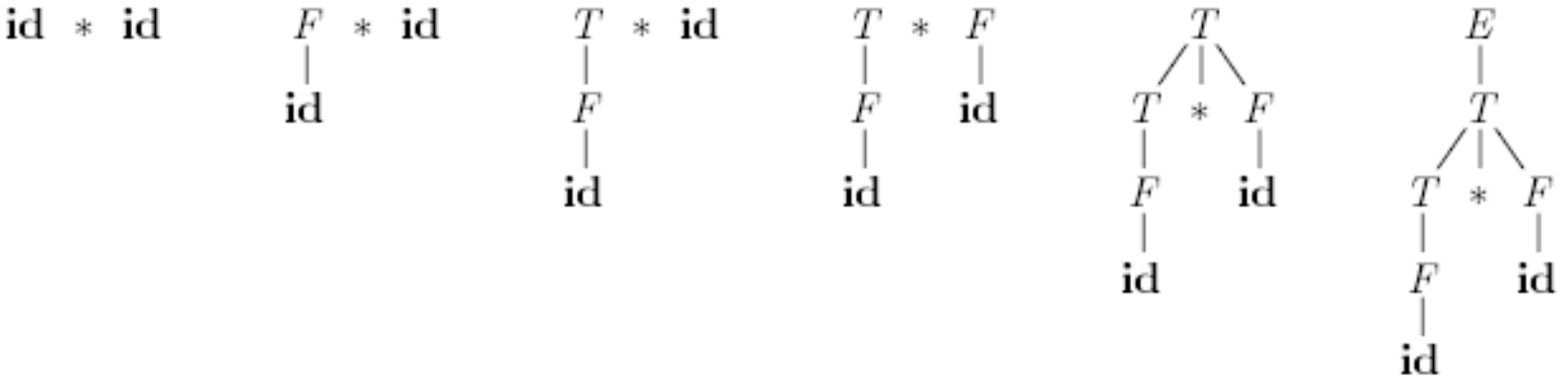

Figure 4.25: A bottom-up parse for **id * id**

Finally…

- We had covered many important idea in the “previous stages” of compilers in this semester
- Compilers is actually a kind of theory of programming language. Though the jobs in the market are quite a few, but the people who knows that is valuable.
- For example, the team for developing MS Visual Studio. Or, the team in JetBrains in developing IntelliJ
- There are still some optimizations we haven’t covered in this semester. You can check the related chapters in the book.