بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice. For example, if we have the two productions

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \end{array}$$

on seeing the input if*, we cannot immediately tell which production to choose to expand* $stmt$*.*

In general, if $A \rightarrow \alpha\beta_1|\alpha\beta_2$ ***are two*** A***-productions, and the input begins with a nonempty string derived from*** α***, we do not know whether to expand*** A ***to*** $\alpha\beta_1$ ***or*** $\alpha\beta_2$***. However, we may defer the decision by expanding*** A ***to*** $\alpha A'$***. Then, after seeing the input derived from*** α***, we expand*** A' ***to*** β_1 ***or to*** β_2***. That is, left-factored, the original productions become***

$$A \rightarrow \alpha A', \qquad A' \rightarrow \beta_1|\beta_2.$$

Algorithm 4.21 : Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Example 4.22: The following grammar abstracts the "dangling-else" problem:

$$\begin{array}{l} S \rightarrow i\ E\ t\ S \ \mid\ i\ E\ t\ S\ e\ S \ \mid\ a \\ E \rightarrow b \end{array} \tag{4.23}$$

Here, i, t, and e stand for **if**, **then**, and **else**; E and S stand for "conditional expression" and "statement." Left-factored, this grammar becomes:

$$\begin{array}{l} S \rightarrow i\ E\ t\ S\ S' \ \mid\ a \\ S' \rightarrow e\ S \ \mid\ \epsilon \\ E \rightarrow b \end{array} \tag{4.24}$$

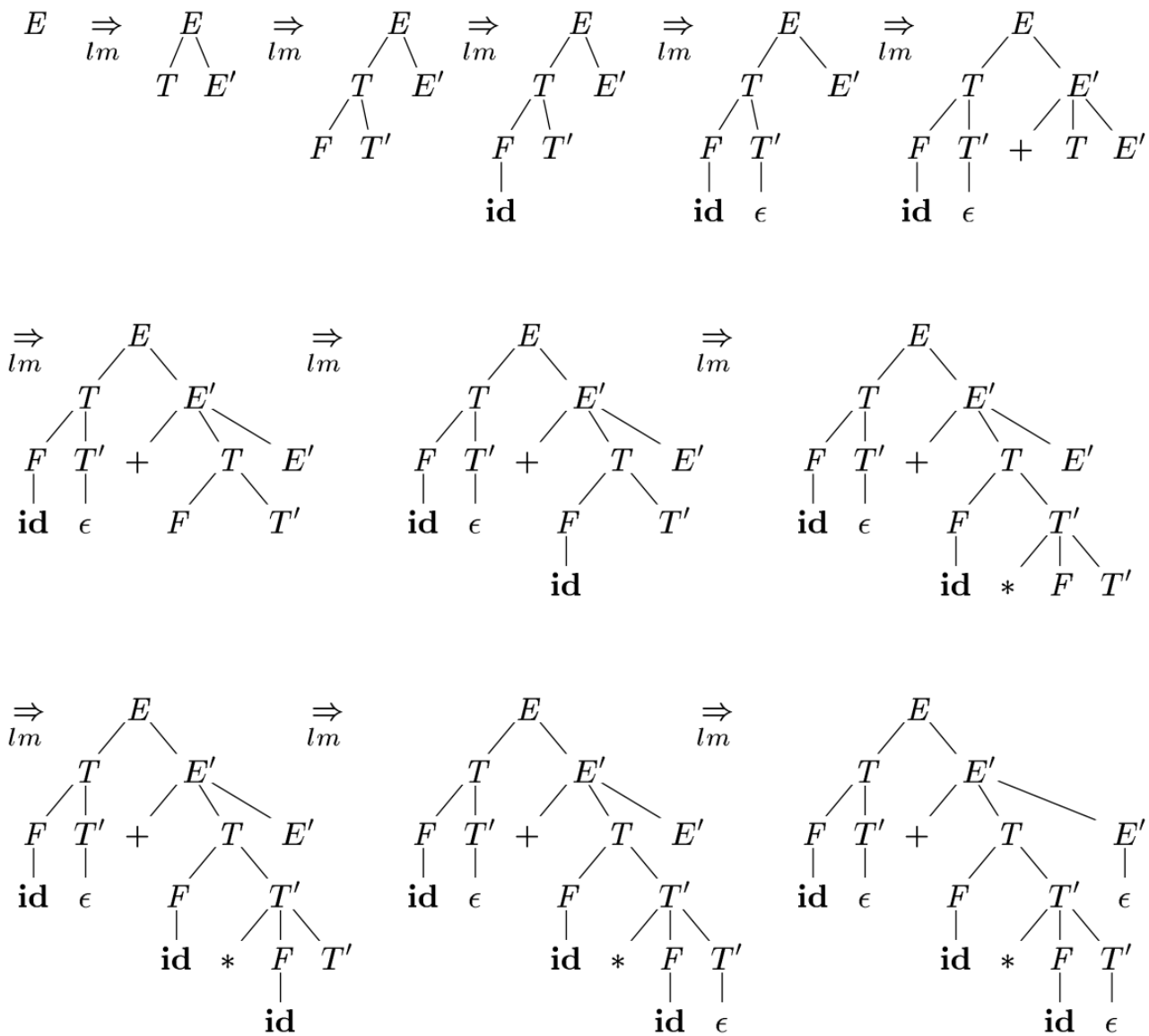
Thus, we may expand S to $iEtSS'$ on input i, and wait until $iEtS$ has been seen to decide whether to expand S' to eS or to ϵ. Of course, these grammars are both ambiguous, and on input e, it will not be clear which alternative for S' should be chosen. Example 4.33 discusses a way out of this dilemma. □

Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example: The sequence of parse trees in the next page for the input $\mathbf{id} + \mathbf{id} * \mathbf{id}$ *is a top-down parse according to the following grammar: This sequence of trees corresponds to a leftmost derivation of the input.*

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
$$


E ⇒lm E T E′ ⇒lm T E′ F T′ ⇒lm F T′ id ⇒lm id ϵ ⇒lm F T′ + T E′
⇒lm ⇒lm id ϵ F T′ id ⇒lm id ϵ F T′ id ∗ F T′
⇒lm ⇒lm id ϵ F T′ id ∗ F T′ id ⇒lm id ϵ F T′ id ∗ F T′ id ϵ

At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say A. Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

Section 4.4 begins with a general form of top-down parsing, called recursive-descent parsing, which may require backtracking to find the correct A-production to be applied.

Predictive parsing: A special case of recursive-descent parsing, where no backtracking is required. Predictive parsing chooses the correct A-production by looking ahead at the input a fixed number of symbols, typically we may look only at one (that is, the next input symbol).

$\mathrm{LL}(k)$ *Class*

The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is sometimes called the $\mathrm{LL}(k)$ class.

we shall see how we can implement $\mathrm{LL}(1)$ parsers as programs. We look at two implementation methods: Recursive descent, where grammar structure is directly translated into the program structure, and a table-based (table-driven) approach that encodes the production choices in a table, so a simple grammar-independent program can use the table to do parsing.

Recursive-Descent Parsing

```
    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

As the name indicates, recursive descent uses recursive functions to implement predictive parsing. The central idea is that each non-terminal in the grammar is implemented by a function in the program.

Each such function looks at the next input symbol in order to choose one of the productions for the nonterminal. The right-hand side of the chosen production is then used for parsing in the following way:

☞ *A terminal on the right-hand side is matched against the next input symbol. If they match, we move on to the following input symbol and the next symbol on the right hand side, otherwise an error is reported.*

☞ *A nonterminal on the right-hand side is handled by calling the corresponding function and, after this call returns, continuing with the next symbol on the right-hand side. When there are no more symbols on the right-hand side, the function returns.*

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

Note that the pseudocode in Fig. 4.13 is nondeterministic, since it begins by choosing the A-production to apply in a manner that is not specified.

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, so backtracking parsers are not seen frequently.

اگر قرار بود کد شکل ۴.۱۳ بک‌ترکینگ داشته باشد:

To allow backtracking, the code of Fig. 4.13 needs to be modified. First, we cannot choose a unique A-production at line (1), so we must try each of several productions in some order. Then, failure at line (7) is not ultimate failure, but suggests only that we need to return to line (1) and try another A-production. Only if there are no more A-productions to try do we declare that an input error has been found. In order to try another A-production, we need to be able to reset the input pointer to where it was when we first reached line (1). Thus, a local variable is needed to store this input pointer for future use.

مثال ۴.۲۹ کتاب را ببینید.

حتی با وجود بک‌ترکینگ هم ممکن است فرایند تجزیه با شکست مواجه شود

A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop. That is, when we try to expand a nonterminal A, we may eventually find ourselves again trying to expand A without having consumed any input.

Lookahead

To recognize P defined as $P \rightarrow Q|R$ we really have to know whether to select the Q route or the R route. If we are to avoid backtracking then there has to be some characteristic of the definitions of Q and R to enable the selection to be made. The way in which this is commonly done is to make use of lookahead.

Suppose the parsing process is at a stage where P has to be recognized. A certain number of tokens have already been consumed from the input. Suppose also that it is possible to examine (without preventing them from being read again) some tokens beyond the current point of input. It may be that the identity of these lookahead tokens allows us to determine whether to follow the Q path or the R path.

For example, if the definition of Q is $Q \rightarrow a \ldots$ and the definition of R is $R \rightarrow b \ldots$ where a and b are terminal symbols, then if the first lookahead token is a we follow the Q path and if the first lookahead token is a b we follow the R path. This idea of lookahead is essential for practical parsers for nontrivial grammars. But managing many tokens of lookahead adds to parser complexity especially if the tokens are being input as they are required. Perhaps surprisingly just a single token lookahead will suffice for the syntax of most programming languages.

The need for lookahead: Writing a predictive recognizing function for $P \rightarrow Q|R$ requires code to determine in all circumstances whether to follow the Q route or the R route. This is done by having one or more tokens in hand and on the basis of the identity of these tokens, the code can decide which route to follow.

Example:

$$S \to Az|z, \qquad A \to xA|B, \qquad B \to y$$

Here, x, y and z are terminal symbols. In the code to recognize an S, if the current lookahead token is a z, we use the production $S \to z$, otherwise (if it is an x or a y) use $S \to Az$. Similarly in the code to recognize an A, if the current lookahead token is a x, we use the production $A \to xA$ and if it is a y use the production $A \to B$. If the current lookahead token is a z this indicates a parse error.

If the lookahead can allow the correct alternative to be chosen in a production involving alternation, as in the example above, then a parser can be written without having to make use of any backtracking. The parsing process has been made deterministic. This allows the construction of a predictive parser.

☞ *The traditional and practical way of coding such a parser is, as we have seen, to associate a function with each non-terminal symbol whose task is to recognize an instance of that non-terminal. These functions call each other according to the syntax rules of the grammar, matching terminal tokens from the input as they go. This is a recursive descent parser. Unfortunately, this approach to parsing will still have difficulties with some particular constructs found in production rules. Fortunately, there are usually uncomplicated solutions.*

☞ *In practice, the favorite top-down technique is the predictive top-down parser, usually with just one token of lookahead (i.e.* $\mathrm{LL}(1)$*). Sometimes greater lookahead can be an advantage, but comes at the cost of increased coding complexity. Furthermore, backtracking top-down parsers are not in general needed or appropriate for programming language parsing.*

نظیر هر متغیر یک تابع داریم

There is a recognizing routine, function or procedure for each of the language's non-terminals. For example, the production rule $P \rightarrow QR$ *is handled by the code* `void P(){ Q(); R();}`
To recognize a P*, a* Q *followed by an* R *have to be recognized. And once recognizing code has been implemented for all the non-terminals in the language, the code recognizing the starting symbol can be called, having the effect of recognizing complete programs written in that language.*
Example:

$$S \rightarrow Az|z, \qquad A \rightarrow xA|B, \qquad B \rightarrow y$$

Here is a complete C program to recognize strings of this language.

```
#include <stdio.h>
#include <stdlib.h>

int ch;

void error(char *msg) {
  printf("Error - found character %c - %s\n",ch,msg);
  exit(1);
}

void b() {
  if (ch == 'y') ch = getchar();
  else error("y expected");
}

void a() {
  if (ch == 'x') {
    ch = getchar();
    a();
  }
  else b();
}
```

```
void s() {
  if (ch == 'z') ch = getchar();
  else {
    a();
    if (ch != 'z') error("z expected");
    else ch = getchar();
  }
  printf("Success!\n");
}


int main(int argc, char *argv[])
{ ch = getchar();
  s();
  return 0;
}
```

Compiling and running this program gives output of the form:

```
$ ./simpletopdown
xyz
Success!
$ ./simpletopdown
xxxxyz
Success!
$ ./simpletopdown
xxxxz
Error - found character z - y expected
$ ./simpletopdown
z
Success!
```

☞ *The program includes a global declaration for a character* `ch`. *As the parser runs this variable always contains the next character from the input—it is the lookahead.*

☞ *In this simple language, all the lexical tokens are single characters and so the lexical analyzer can be replaced by the call to the standard function* `getchar` *to read the next character from the input.*

☞ *Before the parsing process starts by a call to the* `s` *function, the lookahead is initiated by calling* `getchar` *to read the first character of the input.*

☞ *The function* `b` *checks that the current token is a* `y`. *If so, it reads the next input character, but if not, it issues an error message. Here, a function error is called which outputs a message and then halts the execution of the parser.*

☞ *The function* `a` *checks that the current token is an* `x`. *If so, it skips over it and calls a recursively, corresponding to the grammar*

rule $A \rightarrow xA$. Otherwise it calls `b`.

☞ *Finally, the function* `s` *checks for* `z` *and if found, it reads the next token and the parse succeeds. Otherwise,* `a` *is called, and* `s` *checks for a* `z`*. If found, it reads the next token and the parse succeeds again, otherwise an error is generated.*

☞ *The rule to follow is to ensure that on entry to any of the non-terminal recognizing functions, the lookahead variable* `ch` *should already contain the first token of the non-terminal being recognized.*

یک مثال از کتاب توربِن

$$T' \rightarrow T\$$$

$$T \rightarrow R$$

$$T \rightarrow \mathtt{a}T\mathtt{c}$$

$$R \rightarrow$$

$$R \rightarrow \mathtt{b}R$$

Recursive descent parser for the grammar

```
function parseT'() =
  if input = 'a' or input = 'b' or input = '$' then
    parseT() ; match('$')
  else reportError()

function parseT() =
  if input = 'b' or input = 'c' or input = '$' then
    parseR()
  else if input = 'a' then
    match('a') ; parseT() ; match('c')
  else reportError()

function parseR() =
  if input = 'c' or input = '$' then
    (* do nothing, just return *)
  else if input = 'b' then
    match('b') ; parseR()
  else reportError()
```

FIRST، FOLLOW، و *Nullable* را بعداً دقیق معرفی خواهیم کرد

For the `parseT` function, we look at the productions for T. As $FIRST(R) = \{\texttt{b}\}$, the production $T \rightarrow R$ is chosen on the symbol `b`. Since R is also *Nullable*, we must choose this production also on symbols in $FOLLOW(T)$, i.e., `c` or \$. $FIRST(\texttt{a}T\texttt{c}) = \{\texttt{a}\}$, so we select $T \rightarrow \texttt{a}T\texttt{c}$ on an `a`. On all other symbols we report an error.

For `parseR`, we must choose the empty production on symbols in $FOLLOW(R)$ (`c` or \$). The production $R \rightarrow \texttt{b}R$ is chosen on input `b`. Again, all other symbols produce an error.

The above program does not build a syntax tree—it only checks if the input is valid. It can be extended to construct a syntax tree by letting the parse functions return the sub-trees for the parts of input that they parse.

Tree-building recursive descent parser for the grammar

```
function parseT'() =
  if input = 'a' or input = 'b' or input = '$' then
     let tree = parseT() in
        match('$');
        return tree
  else reportError()

function parseT() =
  if input = 'b' or input = 'c' or input = '$' then
    let tree = parseR() in
      return nNode('T', [tree])
  else if input = 'a' then
    match('a') ;
    let tree = parseT() in
      match('c') ;
      return nNode('T', [tNode('a'),tree,tNode('c')])
  else reportError()

function parseR() =
  if input = 'c' or input = '$' then
    return nNode('R', [])
  else if input = 'b' then
    match('b') ;
    let tree = parseR() in
      return nNode('R', [tNode('b'),tree])
  else reportError()
```

Note that, while decisions are made top-down, the syntax tree is built bottom-up by combining sub-trees from recursive calls. We use the functions `tNode` *and* `nNode` *to build nodes in the syntax tree.* `tNode` *takes as argument a terminal symbol and builds a leaf node equal to that terminal.* `nNode` *takes as arguments the name of a nonterminal and a list of subtrees and builds a tree with the nonterminal as root and the subtrees as children. Lists are shown in square brackets with elements separated by commas.*