Compiler Design and Construction
Top-Down Parsing

Slides modified from Louden Book and Dr. Scherger
Top Down Parsing

- A top-down parsing algorithm parses an input string of tokens by tracing out the steps in a *leftmost* derivation.

- Such an algorithm is called top-down because the implied traversal of the parse tree is a preorder traversal.
A top-down parser “discovers” the parse tree by starting at the root (start symbol) and expanding (predict) downward in a depth-first manner.
- They predict the derivation before the matching is done.

A bottom-up parser starts at the leaves (terminals) and determines which production generates them. Then it determines the rules to generate their parents and so-on, until reaching root (S).
Parsing Example

Consider the following Grammar

\[
\begin{align*}
\text{<program>} & \rightarrow \text{begin} \text{<stmts>} \text{end} \ \$ \\
\text{<stmts>} & \rightarrow \text{SimpleStmt} \ ; \ \text{<stmts>} \\
\text{<stmts>} & \rightarrow \text{begin} \ \text{<stmts>} \ \text{end} \ ; \ \text{<stmts>} \\
\text{<stmts>} & \rightarrow \lambda
\end{align*}
\]

• Input:
begin SimpleStmt; SimpleStmt; end $
Top-down Parsing Example

Input: begin SimpleStmt; SimpleStmt; end $

<program>

<program> → begin <stmts> end $
<stmts> → SimpleStmt ; <stmts>
<stmts> → begin <stmts> end ; <stmts>
<stmts> → λ.
Input: begin SimpleStmt; SimpleStmt; end $

<program> → begin <stmts> end $  
<stmts> → SimpleStmt ; <stmts>  
<stmts> → begin <stmts> end ; <stmts>  
<stmts> → λ
Top-down Parsing Example

Input: begin SimpleStmt; SimpleStmt; end $

<program> 

begin <stmts> end $ 

SimpleStmt ; <stmts> 

<program> → begin <stmts> end $ 
<stmts> → SimpleStmt ; <stmts> 
<stmts> → begin <stmts> end ; <stmts> 
<stmts> → λ
Input: begin SimpleStmt; SimpleStmt; end $

<program>
begin <stmts> end $ 
SimpleStmt ; <stmts>
SimpleStmts ; <stmts>

<program> → begin <stmts> end $
<stmts> → SimpleStmt ; <stmts>
<stmts> → begin <stmts> end ; <stmts>
<stmts> → λ.
Top-down Parsing Example

Input: begin SimpleStmt; SimpleStmt; end $

\begin{align*}
<\text{program}> & \rightarrow \text{begin} <\text{stmts}> \text{end} $ \\
<\text{stmts}> & \rightarrow \text{SimpleStmt} ; <\text{stmts}> \\
<\text{stmts}> & \rightarrow \text{begin} <\text{stmts}> \text{end} ; <\text{stmts}> \\
<\text{stmts}> & \rightarrow \lambda
\end{align*}
Two Kinds of Top Down Parsers

- **Predictive parsers** that try to make decisions about the structure of the tree below a node based on a few lookahead tokens (usually one!).
  - This means that only 1 (or k) rules can expand on given terminal.
  - This is a weakness, since little program structure has been seen before predictive decisions must be made.

- **Backtracking parsers** that solve the lookahead problem by backtracking if one decision turns out to be wrong and making a different choice.
  - But such parsers are slow (exponential time in general).
Top Down Parsers (cont.)

Fortunately, many practical techniques have been developed to overcome the predictive lookahead problem, and the version of predictive parsing called recursive-descent is still the method of choice for hand-coding, due to its simplicity.

But because of the inherent weakness of top-down parsing, it is not a good choice for machine-generated parsers. Instead, more powerful bottom-up parsing methods should be used (Chapter 5).
Recursive Descent Parsing

- Simple, elegant idea:
  - Use the grammar rules as recipes for procedure code.
  - Each non-terminal (lhs) corresponds to a procedure.
  - Each appearance of a terminal in the rhs of a rule causes a token to be matched.
  - Each appearance of a non-terminal corresponds to a call of the associated procedure.
Recursive Descent Example

Grammar rule:

\[
    \text{factor} \rightarrow ( \exp ) \mid \text{number}
\]

Code:

```c
void factor(void)
{
    if (token == number) match(number);
    else {
        match('(');
        exp();
        match(')');
    }
}
```
Note how lookahead is not a problem in this example: if the token is *number*, go one way, if the token is ‘(' go the other, and if the token is neither, declare error:

```c
void match(Token expect)
{
    if (token == expect) getToken();
    else error(token, expect);
}
```
A recursive-descent procedure can also compute values or syntax trees:

```c
int factor(void)
{
    if (token == number)
    {
        int temp = atoi(tokStr);
        match(number); return temp;
    }
    else {
        match('('); int temp = exp();
        match(')'); return temp;
    }
}
```
Errors in Recursive Descent Are Tricky to Handle:

- If an error occurs, we must somehow gracefully exit possibly many recursive calls.
  - Best solution: use exception handling to manage stack unwinding (which C doesn’t have!).

- But there are worse problems:
  - left recursion doesn’t work!
Left recursion is impossible!

\[ \text{exp} \rightarrow \text{exp \ addop \ term} \mid \text{term} \]

```c
void exp(void)
{
    if (token == ??)
    {
        exp(); // uh, oh!!
        addop();
        term();
    }
    else term();
}
```
Review on EBNF
Extra Notation:

- **So far: Backus-Naur Form (BNF)**
  - Metasymbols are $| \rightarrow \epsilon$

- **Extended BNF (EBNF):**
  - New metasymbols $[...]$ and $\{\ldots\}$
  - $\epsilon$ largely eliminated by these
EBNF Metasymbols:

- Brackets [...] mean “optional” (like ? in regular expressions):
  \[ \text{exp} \rightarrow \text{term} \ ' | ' \text{exp} | \text{term} \]\  becomes:
  \[ \text{exp} \rightarrow \text{term} [ \ ' | ' \text{exp} ] \]

- if-stmt \rightarrow \text{if} ( \text{exp} ) \text{stmt} 
  \[ \text{if-stmt} \rightarrow \text{if} ( \text{exp} \text{stmt} \text{else stmt} \]
  becomes:
  \[ \text{if-stmt} \rightarrow \text{if} ( \text{exp} ) \text{stmt} [ \text{else stmt} ] \]

- Braces {...} mean “repetition” (like * in regexps - see next slide)
Braces in EBNF

- Replace *only* left-recursive repetition:
  - $exp \rightarrow exp + term \mid term$ becomes:
    - $exp \rightarrow term \{ + term \}$

- Left associativity still implied

- Watch out for choices:
  - $exp \rightarrow exp + term \mid exp - term \mid term$
    - is not the same as
    - $exp \rightarrow term \{ + term \} \mid term \{ - term \}$
Simple Expressions in EBNF

\[
\begin{align*}
\text{exp} & \rightarrow \text{term} \{ \text{addop term} \} \\
\text{addop} & \rightarrow + | - \\
\text{term} & \rightarrow \text{factor} \{ \text{mulop factor} \} \\
\text{mulop} & \rightarrow * \\
\text{factor} & \rightarrow ( \text{exp} ) | \text{number}
\end{align*}
\]
Left recursion is impossible!

\[ \text{exp} \rightarrow \text{exp addop term} \mid \text{term} \]

```c
void exp(void)
{
    if (token == ??)
    {
        exp(); // uh, oh!!
        addop();
        term();
    }
    else term();
}
```
EBNF to the rescue!

\[ \text{exp} \rightarrow \text{term} \{ \text{addop term} \} \]

```
void exp(void)
{
  term();
  while (token is an addop)
  {
    addop();
    term();
  }
}
```
This code can even left associate:

```c
int exp(void)
{
    int temp = term();
    while (token == '+' || token == '-')
    {
        if (token == '+')
        {
            match('+'); temp += term();
        }
        else
        {
            match('-'); temp -= term();
        }
    }
    return temp;
}
```

Left associative tells us that

5-7+2 = ?
-4 or 0
Note that right recursion/assoc. is not a problem:

\[ \text{exp} \rightarrow \text{term} \ [ \text{addop} \ \text{exp} \ ] \]

```c
void exp(void)
{
    term();
    if (token is an addop)
    {
        addop();
        exp();
    }
}
```

Right-associative tells us that
\[ 5 \times 2^2 = ? \]
20 or 100
Or
\[ a = 5; \]
\[ a=b=2 \]
\[ a \ ?= 2 \text{ or } 5 \]
Non-Recursive Top Down Parsing
Step 1: Make DFA-like Transition Diagrams

One can represent the actions of a predictive parser with a transition diagram for each nonterminal of the grammar. For example, let's draw the diagrams for the following grammar:

- $E \rightarrow T \ E'$
- $E' \rightarrow \varepsilon \mid + \ T \ E'$
- $T \rightarrow \ F \ T'$
- $T' \rightarrow \varepsilon \mid \ast \ F \ T'$
- $F \rightarrow \text{id} \mid (E)$

Diagram:

```
E 0 -> T 1 -> E' 2
E' 3 + 4 T 5 E' 6
   \varepsilon 6
T 7 F 8 T' 9
    \ast 10 F 11 T' 12 13
    \varepsilon 13
T' 14 + 15 T 16 E 17 ) 18
    \text{id} 19
F 15 \text{id} 16 E 17 18
```
Top Down Parsing

- To traverse an edge labeled with a nonterminal the parser goes to the starting state of the diagram for that nonterminal and returns to the original diagram when it has reached the end state of that nonterminal.

- The parser has a stack to keep track of these actions.
  - For example, to traverse the $T$-edge from state 0 to state 1, the parser puts state 1 on the top of the stack, traverses the $T$-diagram from state 7 to state 9 and then goes to state 1 after popping it off the stack.
Top Down Parsing

- An edge labeled with a terminal can be traversed when the current input token equals that terminal:
  - When such an edge is traversed, the current input token is replaced with the next input token.
  - For example, the + edge from state 3 to state 4 can be traversed when the parser is in state 3 and the input token is +: traversing the edge will replace the + token with the next token.
An edge labeled with $\varepsilon$ can be traversed if no other edges leaving the current parser state can be traversed:

- The input token remains fixed when an $\varepsilon$-edge is traversed.
- For example, if the parser is in state 3 and the current input token is not a plus sign, $+$, then the parser goes to state 6 and doesn't change the input token.
Let’s optimize to reduce some states

- Notice that after state 5, we have $E'$ again if we see a $T$, so:

Also, $E'$ only shows up in first one so

Combine states
Step 3: Parse

- Create parsing table
  - For each nonterminal, for each input, list next terminal, will have an e-transition as well.
- Inherently recursive so still requires a lot of stack space.
- Optimization to reduce states not always simple
Nonrecursive Predictive Parsing

- Here is a predictive parser that doesn't use recursive descent.

- The program maintains a stack of grammar symbols and uses a two-dimensional M-table created from the grammar.

- A special symbol, $, marks the bottom of the stack and also the end of the input.

- The parser is initialized with the start symbol on the stack and the input pointing to the first token.

![Diagram of Predictive Parsing](image-url)
The actions of the parser depend on the grammar symbol on the top of the stack, $X$, and the current input token, $a$:

- If $X = a = \$\$ then the parser halts and announces successful completion of the parsing.
- If $X = a$ but doesn't equal $\$\$ then the parser pops $X$ off the stack and advances the input to the next token.
- If $X$ is a terminal not equal to $a$ then there is an error.
- If $X$ is a nonterminal then the parser consults entry $M[X, a]$ in the $M$-table.
- If the $M[X, a]$ entry is a production for $X$ then the parser pops $X$ off the stack and pushes the symbols on the right-side of the production onto the stack (pushing the rightmost symbol of the right-side first and pushing the leftmost symbol on the right-side last.)
- If the $M[X, a]$ is an error entry then the parser announces the error and calls an error recovery routine.
Nonrecursive Predictive Parsing

- So given the following grammar... its corresponding M-Table is

\[
\begin{align*}
E & \rightarrow T E' \\
E' & \rightarrow \varepsilon \mid + T E' \\
T & \rightarrow FT' \\
T' & \rightarrow \varepsilon \mid * FT' \\
F & \rightarrow \text{id} \mid (E)
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{Nonterminal} & \text{Input Symbol} \\
\hline
E & id & + & * & ( & \varepsilon & \varepsilon & \text{(E)} \\
\hline
E' & E' \rightarrow TE' & & E' \rightarrow \varepsilon & & \varepsilon & \\
\hline
T & T \rightarrow FT' & & & T-FT' & & \\
\hline
T' & T' \rightarrow \varepsilon & T' \rightarrow FT' & & T' \rightarrow \varepsilon & & T' \rightarrow \varepsilon \\
\hline
F & F \rightarrow \text{id} & & & & F \rightarrow (E) & \\
\hline
\end{array}
\]
Nonrecursive Predictive Parsing

- Using our grammar and M-Table, show the stack moves made by the predictive parser on input \texttt{id+id*id}

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id+id*id</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>E</td>
<td>E $\rightarrow$ TE'</td>
<td>E $\rightarrow$ TE'</td>
</tr>
<tr>
<td>E'</td>
<td>E $\rightarrow$ +TE'</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T $\rightarrow$ FT'</td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>T' $\rightarrow$ ε</td>
<td>T' $\rightarrow$ *FT'</td>
</tr>
<tr>
<td>F</td>
<td>F $\rightarrow$ id</td>
<td></td>
</tr>
</tbody>
</table>
Nonrecursive Predictive Parsing

- Using our grammar and M-Table, show the stack moves made by the predictive parser on input \texttt{id+id*id}

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id+id*id</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>E</td>
<td>$E \rightarrow TE'$</td>
<td>E$E' \rightarrow TE'$</td>
</tr>
<tr>
<td>E'</td>
<td>E'$ \rightarrow +TE'$</td>
<td>E'$ \rightarrow \epsilon$</td>
</tr>
<tr>
<td>T</td>
<td>T$ \rightarrow FT'$</td>
<td>T$ \rightarrow FT'$</td>
</tr>
<tr>
<td>T'</td>
<td>T'$ \rightarrow \epsilon$</td>
<td>T'$ \rightarrow *FT'$</td>
</tr>
<tr>
<td>F</td>
<td>F$ \rightarrow id$</td>
<td>F$ \rightarrow (E)$</td>
</tr>
</tbody>
</table>
Nonrecursive Predictive Parsing

- Using our grammar and M-Table, show the stack moves made by the predictive parser on input **id+id*id**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id+id*id</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>$E$</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$E' \rightarrow +TE'$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$T'$</td>
<td>$T' \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow id$</td>
</tr>
</tbody>
</table>
### Nonrecursive Predictive Parsing

- Using our grammar and M-Table, show the stack moves made by the predictive parser on input **id+id*id**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E'T$</td>
<td><strong>id+id*id</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>E</td>
<td>$E \rightarrow TE'$</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td>E'</td>
<td>$E' \rightarrow +TE'$</td>
<td>$E' \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>T</td>
<td>$T \rightarrow FT'$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>T'</td>
<td>$T' \rightarrow \varepsilon$</td>
<td>$T' \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>F</td>
<td>$F \rightarrow id$</td>
<td>$F \rightarrow (E)$</td>
</tr>
</tbody>
</table>
Nonrecursive Predictive Parsing

Using our grammar and M-Table, show the stack moves made by the predictive parser on input $id+id*id$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E’T$</td>
<td>$id+id*id$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
<th>Nonterminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>E</td>
<td>$\rightarrow$ TE’</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>E</td>
<td>$\rightarrow$ TE’</td>
</tr>
<tr>
<td>E’</td>
<td>(</td>
<td>E’</td>
<td>$\rightarrow$ $\epsilon$</td>
</tr>
<tr>
<td></td>
<td>)</td>
<td>E’</td>
<td>$\rightarrow$ $\epsilon$</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T’</td>
<td>$\rightarrow$ $\epsilon$</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$ FT'</td>
<td>T’</td>
<td>$\rightarrow$ $\epsilon$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T’</td>
<td>$\rightarrow$ $\epsilon$</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>$\rightarrow$ (E)</td>
</tr>
</tbody>
</table>
Nonrecursive Predictive Parsing

- Using our grammar and M-Table, show the stack moves made by the predictive parser on input \textit{id+id*id}

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E'T'F$</td>
<td>id+id*id</td>
<td></td>
</tr>
</tbody>
</table>

Nonterminal | Input Symbol | Input Symbol |
-------------|--------------|--------------|
| id          | +            | *            |
| E           | $E\rightarrow TE'$ | $E\rightarrow TE'$ |
| E'          | $E'\rightarrow +TE'$ | $E'\rightarrow \varepsilon$ | $E'\rightarrow \varepsilon$ |
| T           | $T\rightarrow FT'$ | T-FT' |
| T'          | $T'\rightarrow \varepsilon$ | $T'\rightarrow *FT'$ | $T'\rightarrow \varepsilon$ | $T'\rightarrow \varepsilon$ |
| F           | $F\rightarrow id$ | $F\rightarrow (E)$ |
Nonrecursive Predictive Parsing

- Using our grammar and M-Table, show the stack moves made by the predictive parser on input id+id*id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E'T'id$</td>
<td>id+id*id</td>
<td></td>
</tr>
</tbody>
</table>

Using our grammar and M-Table, show the stack moves made by the predictive parser on input id+id*id

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>E</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td>E'</td>
<td>$E' \rightarrow +TE'$</td>
</tr>
<tr>
<td>T</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>T'</td>
<td>$T' \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>F</td>
<td>$F \rightarrow id$</td>
</tr>
</tbody>
</table>
Nonrecursive Predictive Parsing

- Using our grammar and M-Table, show the stack moves made by the predictive parser on input \text{id+id*id}

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{E’T’id}$</td>
<td>\text{id+id*id}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>E</td>
<td>E$\rightarrow$TE'</td>
<td>$\rightarrow$TE'</td>
</tr>
<tr>
<td>E'</td>
<td>E'$\rightarrow$TE'</td>
<td>E'$\rightarrow$ε</td>
</tr>
<tr>
<td>T</td>
<td>T$\rightarrow$FT'</td>
<td>T-FT'</td>
</tr>
<tr>
<td>T'</td>
<td>T'$\rightarrow$ε</td>
<td>T'$\rightarrow$FT'</td>
</tr>
<tr>
<td>F</td>
<td>F$\rightarrow$id</td>
<td>F$\rightarrow$(E)</td>
</tr>
</tbody>
</table>
Nonrecursive Predictive Parsing

- Using our grammar and M-Table, show the stack moves made by the predictive parser on input \texttt{id+id*id}

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E'T'$</td>
<td>+id*id</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>*</td>
</tr>
<tr>
<td>E</td>
<td>$E \rightarrow TE'$</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td>E'</td>
<td>$E' \rightarrow +TE'$</td>
<td>$E' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>T</td>
<td>$T \rightarrow FT'$</td>
<td>$T-FT'$</td>
</tr>
<tr>
<td>T'</td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow *FT'$</td>
</tr>
<tr>
<td>F</td>
<td>$F \rightarrow id$</td>
<td>$F \rightarrow (E)$</td>
</tr>
</tbody>
</table>
Parse for \( \text{id+id*id} \)

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E )</td>
<td>( E \rightarrow TE' )</td>
</tr>
<tr>
<td>( E' )</td>
<td>( E' \rightarrow +TE' )</td>
</tr>
<tr>
<td>( T )</td>
<td>( T \rightarrow FT' )</td>
</tr>
<tr>
<td>( T' )</td>
<td>( T' \rightarrow \epsilon )</td>
</tr>
<tr>
<td>( F )</td>
<td>( F \rightarrow id )</td>
</tr>
</tbody>
</table>

### Stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id$id$</td>
<td>( E \rightarrow TE' )</td>
</tr>
<tr>
<td>$ET$</td>
<td>id$id$</td>
<td>( E \rightarrow TE' )</td>
</tr>
<tr>
<td>$ETF$</td>
<td>id$id$</td>
<td>( T \rightarrow FT' )</td>
</tr>
<tr>
<td>$ET'id$</td>
<td>id$id$</td>
<td>( F \rightarrow id )</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$id$id$</td>
<td>( F \rightarrow id )</td>
</tr>
<tr>
<td>$E'$</td>
<td>$id$id$</td>
<td>( T' \rightarrow \epsilon )</td>
</tr>
<tr>
<td>$E'T'+$</td>
<td>$id$id$</td>
<td>( E' \rightarrow +TE' )</td>
</tr>
<tr>
<td>$E'T$</td>
<td>id$id$</td>
<td>( E' \rightarrow +TE' )</td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id$id$</td>
<td>( T \rightarrow FT' )</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$id$</td>
<td>( F \rightarrow id )</td>
</tr>
<tr>
<td>$E'T'F*$</td>
<td>$id$</td>
<td>( T' \rightarrow *FT' )</td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>$id$</td>
<td>( F \rightarrow id )</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>$id$</td>
<td>( F \rightarrow id )</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>
First and Follow

- FIRST and FOLLOW are two functions associated with a grammar that help us fill in the entries of an M-table. The functions have other uses as well.

- If $Z$ is any string of grammar symbols then $\text{FIRST}(Z)$ is the set of all terminals that begin strings derived from $Z$.
  - That is Terminals that can start a valid string generated by $Z$

- If $Z \Rightarrow^* \varepsilon$ then $\varepsilon$ is also in $\text{FIRST}(Z)$.
If \( A \) is a nonterminal then \( \text{FOLLOW}(A) \) is the set of all terminals that can appear immediately after \( A \) in some sentential form derived from the start symbol.

- Set of terminals that can follow \( A \) in some legal derivation

If \( A \) appears as the rightmost symbol in some sentential form then the end of input, $, is also in \( \text{FOLLOW}(A) \).
First and Follow

To compute FIRST(X)

1. If X is a terminal, then FIRST(X) is \{X\}
2. If X \rightarrow \varepsilon is a production, then add \varepsilon to FIRST(X)
3. If X is a nonterminal and X \rightarrow Y_1 Y_2 \ldots Y_k is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and \varepsilon is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon.
4. If \varepsilon is in FIRST(Y_j) for all j = 1, 2, ..., k, then add \varepsilon to FIRST(X). If Y_1 does not derive \varepsilon, then we add nothing more to FIRST(X), but if Y_1 \Rightarrow^* \varepsilon, then we add FIRST(Y_2) and so on.
First and Follow

To compute FOLLOW(X)

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right endmarker
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for $\epsilon$ is placed in FOLLOW(B)
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains $\epsilon$ (i.e., $\beta \Rightarrow^{*} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B)
First and Follow

- So for our grammar

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow \varepsilon | + TE' \\
T & \rightarrow FT' \\
T' & \rightarrow \varepsilon | * FT' \\
F & \rightarrow \text{id} | (E) \\
\end{align*}
\]

id and (are added to FIRST(F) by rule 3; i=1 in each case, since \( \text{FIRST(id)} = \{\text{id}\} \) and \( \text{FIRST('')} = \{()\} \) by rule 1.

Using rule 3 with i=1, \( T \rightarrow FT' \) implies that \text{id} and ( belong to FIRST(T)

1. If \( X \) is a terminal, then FIRST(X) is \{X\}
2. If \( X \rightarrow \varepsilon \) is a production, then add \( \varepsilon \) to FIRST(X)
3. If \( X \) is a nonterminal and \( X \rightarrow Y_1 Y_2 \ldots Y_k \) is a production, then place \( a \) in FIRST(X) if for some \( i \), \( a \) is in FIRST\( (Y_i) \), and \( \varepsilon \) is in all of FIRST\( (Y_1), \ldots, \text{FIRST}(Y_{i-1}) \); that is, \( Y_1 \ldots Y_{i-1} \Rightarrow^{*} \varepsilon \).
4. If \( \varepsilon \) is in FIRST\( (Y_j) \) for all \( j = 1, 2, \ldots, k \), then add \( \varepsilon \) to FIRST(X). If \( Y_1 \) does not derive \( \varepsilon \), then we add nothing more to FIRST(X), but if \( Y_1 \Rightarrow^{*} \varepsilon \), then we add FIRST\( (Y_2) \) and so on.
First and Follow

- So for our grammar

\[
E \rightarrow TE' \\
E' \rightarrow \varepsilon | + TE' \\
T \rightarrow FT' \\
T' \rightarrow \varepsilon | * FT' \\
F \rightarrow id | (E)
\]

\[
\begin{array}{|c|c|c|}
\hline
A & FIRST & FOLLOW \\
\hline
E & (id) & )$ \\
E' & + \varepsilon & )$ \\
T & (id) & +)$ \\
T' & * \varepsilon & +)$ \\
F & (id) & +*)$ \\
\hline
\end{array}
\]

First(E) = First(T) = {id,\(\)}
First(E') = {+,\(\)}
First(T') = {\(\),*}
First and Follow

- So for our grammar
  
  \[ E \rightarrow TE' \]
  
  \[ E' \rightarrow \epsilon | + TE' \]
  
  \[ T \rightarrow FT' \]
  
  \[ T' \rightarrow \epsilon | * FT' \]
  
  \[ F \rightarrow \text{id} | (E) \]

  put $ in Follow(E) by rule 1.
  
  rule 2 on F-$(E), add ) is to Follow(E).
  
  Apply rule 3 to E-$(TE', $ and ) are in Follow(E').

1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right endmarker.
2. If there is a production \( A \rightarrow \alpha B \beta \), then everything in FIRST(\( \beta \)) except for \( \epsilon \) is placed in FOLLOW(\( B \)).
3. If there is a production \( A \rightarrow \alpha B \), or a production \( A \rightarrow \alpha B \beta \) where FIRST(\( \beta \)) contains \( \epsilon \) (i.e., \( \beta \rightarrow^* \epsilon \)), then everything in FOLLOW(A) is in FOLLOW(B).
First and Follow

- So for our grammar

\[
E \rightarrow TE' \\
E' \rightarrow \varepsilon | + TE' \\
T \rightarrow FT' \\
T' \rightarrow \varepsilon | * FT' \\
F \rightarrow \text{id} | (E) \\
\]

Follow(E) = Follow(E') = \{,\},$
Follow(T) = Follow(T') =\{+,\},$
Follow(F) =\{+,*,\},$

<table>
<thead>
<tr>
<th>A</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>( id</td>
<td>) $</td>
</tr>
<tr>
<td>E'</td>
<td>+ \varepsilon</td>
<td>) $</td>
</tr>
<tr>
<td>T</td>
<td>( id</td>
<td>+ ) $</td>
</tr>
<tr>
<td>T'</td>
<td>* \varepsilon</td>
<td>+ ) $</td>
</tr>
<tr>
<td>F</td>
<td>( id</td>
<td>+ * ) $</td>
</tr>
</tbody>
</table>
Another Example

S → +SS | *SS | a;

FIRST(S) = {+, *, a}
FOLLOW(S) = {+, *, a, $}
Construction of Predictive Parsing Tables

- INPUT: Grammar G
- OUTPUT: Parsing Table M
- Method:
  - For each production \( A \rightarrow \alpha \) of the grammar, do steps 2 and 3
  - For each terminal \( a \) in \( \text{FIRST}(\alpha) \), add \( A \rightarrow \alpha \) to \( M[A,a] \)
  - If \( \epsilon \) is in \( \text{FIRST}(\alpha) \), add \( A \rightarrow \alpha \) to \( M[A,b] \) for each terminal \( b \) in \( \text{FOLLOW}(A) \). If \( \epsilon \) is in \( \text{FIRST}(\alpha) \) and $ is in \( \text{FOLLOW}(A) \), add \( A \rightarrow \alpha \) to \( M[A,\$] \)
  - Make each undefined entry of \( M \) be error

\[
\text{Predict}(A \rightarrow X_1 \ldots X_m) =
\begin{align*}
& (\text{First}(X_1 \ldots X_m) - \epsilon) \cup \text{Follow}(A) & \text{if } \epsilon \in \text{First}(X_1 \ldots X_m) \\
& \text{First}(X_1 \ldots X_m) & \text{otherwise}
\end{align*}
\]
LL(1) Grammars

- Should the previous algorithm put two or more different productions in the same entry of the M-table it means that the grammar is ambiguous and/or left-recursive and/or not left-factored.

- A grammar is an $LL(1)$-grammar if and only if its M-table has no entries that are multiply-defined.
LL(1) Grammars

- So for the following grammar

```
stmt --> a | if expr then stmt opt_else
opt_else --> ε | else stmt
expr --> b
```

- S → a | i E t S S'
- S' → ε | e S
- E → b

- There are two productions in the M(opt_else, else) entry so the grammar is ambiguous.
- To resolve the ambiguity we must delete either the opt_else --> else stmt production or the opt_else --> ε production from this entry.
- Since the opt_else --> else stmt production is the only production in the grammar that handles the else token we must keep it and drop the opt_else --> ε production from this entry.
- This choice corresponds with associating else tokens with the closest previous unmatched then tokens.

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td>S</td>
<td>S→a</td>
</tr>
<tr>
<td>S'</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E→b</td>
</tr>
</tbody>
</table>
Another Example

- $S \rightarrow +SS \mid *SS \mid a;$

- $\text{FIRST}(S) = \{+, *, a\}$
- $\text{FOLLOW}(S) = \{+, *, a, $\}$ '

- Parse Table M

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Nonterminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>S $\rightarrow$ a</td>
</tr>
<tr>
<td>$+$</td>
<td>S $\rightarrow$ +SS</td>
</tr>
<tr>
<td>*</td>
<td>S $\rightarrow$ *SS</td>
</tr>
<tr>
<td>$$</td>
<td>error</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input Symbol</th>
<th>Nonterminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>S $\rightarrow$ a</td>
</tr>
<tr>
<td>$+$</td>
<td>S $\rightarrow$ +SS</td>
</tr>
<tr>
<td>*</td>
<td>S $\rightarrow$ *SS</td>
</tr>
<tr>
<td>$$</td>
<td>error</td>
</tr>
</tbody>
</table>
Another Example

- $S \rightarrow ( S ) S \mid \varepsilon$

- $\text{FIRST}(S) = \{(, \varepsilon\}$
- $\text{FOLLOW}(S) = \{\}, \$

- Parse Table M

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$S \rightarrow (S)S$</td>
</tr>
</tbody>
</table>
Another Example

- $S \rightarrow S \ ( \ S \ ) \ | \ \varepsilon$

- $\text{FIRST}(S) = \{ (, \ \varepsilon \}$
- $\text{FOLLOW}(S) = \{ (, \ ), \ \$\}$

- Parse Table M

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(</td>
</tr>
<tr>
<td>$S$</td>
<td>$S \rightarrow (S)S$</td>
</tr>
</tbody>
</table>
Error Recovery in Predictive Parsing

- The stack of a nonrecursive predictive parser shows what the parser hopes to match with the remainder of the input.

- The parser detects an error whenever there is a terminal on the top of the stack that doesn't agree with the current input token or when it consults an M-table entry marking an error.

- The FIRST and FOLLOW sets of a grammar can be used to generate meaningful error messages and expedite error recovery.
Here are five heuristics one can use.

(1) As a starting point, we can place all symbols in $\text{FOLLOW}(A)$ into the synchronizing set for nonterminal $A$. If we skip tokens until an element of $\text{FOLLOW}(A)$ is seen and pop $A$ from the stack, it is likely that parsing can continue.

(2) If is not enough to use $\text{FOLLOW}(A)$ as the synchronizing set for $A$. For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the $\text{FOLLOW}$ set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchical structure on constructs in a language. E.g. expressions appear within statements, which appear within blocks, and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.
Error Recovery in Predictive Parsing

- Here are five heuristics one can use.

- (3) If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

- (4) If a nonterminal can generate the empty string, then the production deriving $\varepsilon$ can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.

- (5) If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.
Syntax Directed Definitions

- A syntax-directed definition generalizes a context-free grammar by associating a set of attributes which each node in a parse tree.

- Each attribute gives some information about the node.
  - For example, attributes associated with an expression-node may give its value, its type, or its location in memory, etc.

- There are two kinds of attributes:
  - The value of a synthesized attribute at a node depends on attribute values at the node's children.
  - The value of an inherited attribute at a node depends on attribute values at its parent node and/or its sibling nodes.
Since the root of a parse tree has no parent and no siblings, the start symbol of a grammar can have no inherited attributes.

Information about terminal symbols at the leaves of a parse tree comes from the lexical analyzer (or in a field of a symbol table entry that the lexical analyzer points to) and we treat this information as synthesized.

A parse tree showing the values of attributes at each node is called an *annotated* parse tree:

- computing the attribute values is called *annotating* or *decorating* the tree.
Semantic rules are associated with the productions of the grammar to show the relationships between the attributes of each parent node and its children nodes.

For example, assume there is a production in a grammar,

\[ X \rightarrow Y \ Z \]

that constructs a parse tree with nodes Y and Z as children of node X and further assume there is an attribute, \( a \), attached to each of the nodes as shown.
If there is a semantic rule, \( X.a := f(Y.a, Z.a) \), associated with production \( X \rightarrow Y Z \) then attribute \( X.a \) of the parent node is a \textbf{synthesized} attribute which can be evaluated by applying function \( f \) to attributes \( Y.a \) and \( Z.a \) of its children.

On the other hand, if there is a semantic rule, \( Z.a := f(X.a, Y.a) \), associated with production \( X \rightarrow Y Z \) then attribute \( Z.a \) of the right child is an \textbf{inherited} attribute which can be evaluated by applying function \( f \) to attributes \( X.a \) and \( Y.a \) of its parent and sibling.
Here is the syntax-directed definition of a simple desk calculator.

In this example, the \textit{val} attribute of every node is a synthesized attribute.

Note the use of subscripts in a production like \( E \rightarrow E + T \) where the same grammar symbol appears more than once:

- the \( E \) child node is given a subscript of 1 to distinguish it from the \( E \) parent node (in the production and in the associated semantic rule.)

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L \rightarrow En )</td>
<td>Print(E.val)</td>
</tr>
<tr>
<td>( E \rightarrow E_1 + T )</td>
<td>( E.\text{Val} := E_1.\text{val} + T.\text{val} )</td>
</tr>
<tr>
<td>( E \rightarrow T )</td>
<td>( E.\text{Val} := T.\text{val} )</td>
</tr>
<tr>
<td>( T \rightarrow T_1 * F )</td>
<td>( T.\text{Val} := T_1.\text{val} * F.\text{val} )</td>
</tr>
<tr>
<td>( T \rightarrow F )</td>
<td>( T.\text{Val} := F.\text{val} )</td>
</tr>
<tr>
<td>( F \rightarrow (E) )</td>
<td>( F.\text{Val} := E.\text{val} )</td>
</tr>
<tr>
<td>( F \rightarrow \text{digit} )</td>
<td>( F.\text{Val} := \text{digit lexval} )</td>
</tr>
</tbody>
</table>
S-attributed Functions

- A syntax-directed definition is an *S-attributed definition* if all attributes are synthesized.

- The preceding table is an example of an S-attributed definition.

- A parse tree for an S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node bottom-up, from the leaves to the root.
Inherited Attributes

- Inherited attributes are useful for passing type information in declarations.

- Let's derive a syntax-directed definition for a declaration in C.

- It uses a synthesized attribute, $T.type$, to collect the type of the declaration and an inherited attribute, $L.in$, to pass the type down through the list of id nodes in the declaration so their symbol table entries can be updated.
Construction of Syntax Trees

- A syntax tree or an abstract syntax tree (AST) is a condensed form of a parse tree with the operators and keywords associated with interior nodes rather than with the leaves.
- For example, the production: \( stmt \rightarrow if \ expr \ then \ stmt \) appears in a syntax tree like:

```
if-then

  expr
  statement
```

```
if-then

  if

  expr

  then

  statement
```
Construction of Syntax Trees

- As another example consider the parse tree constructed for $9 - 5 + 2$ in our course notes.
- The syntax tree for this expression is simply:
Construction of Syntax Trees

- Here is the syntax-directed definition for constructing a syntax tree for an expression...

- Attribute $nptr$ is a pointer to a node of the syntax tree. When function $mknodex$ is given an operator and pointers to two nodes it creates a parent node for those two nodes labeled with the operator and returns a pointer to the node it creates. Similarly, function $mkleafx$ creates a leaf and returns a pointer to it.
L-Attributed Definitions

• In general, an inherited attribute of a node depends on attributes of its parent node and on attributes of its sibling nodes.

• It is often the case where an inherited attribute of a node depends only on the inherited attributes of its parent node and on attributes of sibling nodes to its left:
  • i.e., there is no dependence on a synthesized attribute of the parent nor on any attribute of a sibling node on the right.

• If this is true of all inherited attributes in a syntax-directed definition then it is \textit{L-attributed}.
  • Note that there is no restriction on the synthesized attributes of the definition; e.g., every S-attributed definition is also L-attributed.
Recursive Procedure for DFVisit

procedure dfvisit(n : node);
Begin
  for each child m of n in left-to-right order do
    Begin
      evaluate inherited attributes of m ;
      dfvisit(m );
    end; {for loop}
  evaluate synthesized attributes of n ;
end

• Calling dfvisit at the root of a parse tree for an L-attributed definition will annotate the whole parse tree.
Translation Schemes

• Translation schemes are introduced earlier

• A translation scheme is a context-free grammar (with attributes associated with the grammar symbols) where semantic actions (enclosed in braces) are inserted within the right-sides of productions.

• We have looked at a translation scheme for printing an infix expression in postfix notation.
Translation Schemes

• A translation scheme is a convenient way of describing an L-attributed definition.

• As an example, assume the grammar has a production: $A \rightarrow X \ Y$ and further assume that $A$, $X$, and $Y$, have inherited attributes $A.i$, $X.i$, and $Y.i$, and synthesized attributes $A.s$, $X.s$, and $Y.s$, respectively.

• Because we have an L-attributed definition:
  • $X.i$ can only be a function of $A.i$; e.g., $X.i := f(A.i)$;
  • $Y.i$ can only be a function of $A.i$, $X.i$, and $X.s$; e.g., $Y.i := g(A.i, X.i, X.s)$; and
  • $A.s$ is a function of $A.i$, $X.i$, $X.s$, $X.i$, and $X.s$; e.g., $A.s := h(A.i, X.i, X.s, Y.i, Y.s)$. 
Translation Schemes

• A translation scheme would embed the following semantic actions in the production $A \rightarrow X Y$ as follows:
  
  $A \rightarrow$  
  
  $\{$  
  $X.i := f(A.i);$ \}  
  $X$  
  $\{$  
  $Y.i := g(A.i, X.i, X.s);$ \}  
  $Y$  
  $\{$  
  $A.s := h(A.i, X.i, X.s, Y.i, Y.s);$ \}$

• Note the careful placement of the semantic actions in the production:
  
  • if any semantic action is moved later in the production then an inherited attribute of a child won't be evaluated in time and if any action is moved earlier in the production it will try to use an argument that hasn't been evaluated.

• There is no special problem with $\varepsilon$-productions in the grammar.

• For example, assume $A \rightarrow \varepsilon$ is a production in the grammar and assume that $A$ has an inherited attribute, $A.i$, and a synthesized attribute $A.s$, that is a function, $f$, of $A.i$.

• Then the translation scheme contains:
  
  $A \rightarrow$  
  
  $\{$  
  $A.s := f(A.i);$ \}$
Top Down Translation

• This section describes how L-attributed definitions can be implemented with predictive parsers.

• Translation schemes are used instead of syntax-directed definitions so the order in which semantic actions and attribute evaluations should occur is shown explicitly.
Eliminating Left Recursion From a Translation Scheme

• Most arithmetic operators are left-associative so it is natural to use left-recursive grammars for expressions: also there are other language constructs best described with left-recursive grammars.

• But left recursion must be eliminated before a predictive parser can parse a grammar.

• What do we do when the grammar of a translation scheme is left-recursive?

• Can every semantic action and attribute evaluation of a translation scheme be put in its proper place when we eliminate left recursion from its grammar?
Example

- A left-recursive grammar for a list of digits separated by plus and minus signs is shown below. The parse tree for $9 - 5 + 2$ is also shown:
Example

• Note the chain of $E$-nodes going down toward the left from the root of the parse tree. Addition and subtraction are left-associative so to evaluate $9 - 5 + 2$ properly we should go through the chain of $E$-nodes from the bottom up to the root. A translation scheme needs only a synthesized attribute ($\text{val}$) to properly evaluate a list of digits separated by plus and minus signs:

\[
\begin{align*}
E &\to E1 + T \{ E.\text{val} := E1.\text{val} + T.\text{val} \} \\
E &\to E1 - T \{ E.\text{val} := E1.\text{val} - T.\text{val} \} \\
E &\to T \{ E.\text{val} := T.\text{val} \} \\
T &\to 0 \{ T.\text{val} := 0 \} \\
\ldots \\
T &\to 9 \{ T.\text{val} := 9 \}
\end{align*}
\]
Example

• Eliminating left recursion from the grammar shown above produces the grammar shown below.
• The parse tree for 9 - 5 + 2 with this new grammar is also shown below:
Example

- Note that the new parse tree has a chain of $R$-nodes going down toward the right from its root whereas the first parse tree has a chain of $E$-nodes going down toward the left from its root.

- Addition and subtraction are still left-associative so to properly evaluate $9 - 5 + 2$ we must now go down through the chain of $R$-nodes from the root toward the $R \rightarrow \varepsilon$ node at the bottom.
Translation Schemes

- A translation scheme with this new grammar needs an inherited attribute (\(in\)) to properly evaluate a list of digits separated by plus and minus signs and the scheme sends the final result into the \(R \rightarrow \varepsilon\) node at the bottom of the chain.

- The final result should really be sent to the root of the parse tree so the translation scheme also needs a synthesized attribute (\(syn\)) to move the final result from the \(R \rightarrow \varepsilon\) node back up the chain of \(R\)-nodes:

\[
\begin{align*}
E \rightarrow & \ T \ {\{R.in := T.val\}} \ R \ {\{E.val := R.syn\}} \\
R \rightarrow & \ + \ T \ {\{R1.in := R.in + T.val\}} \ R1 \ {\{R.syn := R1.syn\}} \\
R \rightarrow & \ - \ T \ {\{R1.in := R.in - T.val\}} \ R1 \ {\{R.syn := R1.syn\}} \\
R \rightarrow & \ \varepsilon \ {\{R.syn := R.in\}} \\
T \rightarrow & \ 0 \ {\{T.val := 0\}} \\
\ldots & \\
T \rightarrow & \ 9 \ {\{T.val := 9\}}
\end{align*}
\]
Translation Schemes

- **General Case:** In general, there may be both right-associative operators and left-associative operators in a translation scheme.
- Right-associative operators pose no problem because they don't introduce left recursion.
- Left-associative operators make the scheme left-recursive but the left recursion can be easily eliminated from the grammar using the algorithms shown here.
- Eliminating the left recursion changes parse trees by replacing each chain of nodes going down toward the left with a chain of nodes going down to the right.
- Each synthesized attribute that was originally evaluated going up the original chain is replaced by an inherited attribute that is evaluated going down the new chain.
- The result of the evaluation can be sent back up the new chain with another synthesized attribute.
Design of A Predictive Translator

- A parse tree of any L-attributed definition can be completely annotated by calling the recursive \texttt{dfvisit} procedure for the root of the tree.
- The construction of a recursive-descent predictive parser is described earlier.
- Note that the flow of control through \texttt{dfvisit} is similar to the flow of control through a recursive-descent predictive parser:
  - control flows into a node from its parent, flows in and out of each of its children (from left-to-right) and then returns to the parent.
- In \texttt{dfvisit} the inherited attributes of each node are evaluated before the node is visited and the synthesized attributes are evaluated just before control returns to the parent of the node.
Design of A Predictive Translator

- Changing a recursive-descent predictive parser into a predictive translator is simple:
  - Evaluate the inherited attributes of a nonterminal before calling the recursive procedure for that nonterminal.
  - Pass the values of these inherited attributes into the procedure as arguments in the call.
  - The procedure for each nonterminal evaluates it synthesized attributes before returning to its caller.
  - Pass the values of synthesized attributes back to the caller as returned values.
Error Recovery in Parsers

- A parser should try to determine that an error has occurred as soon as possible. Waiting too long before declaring error means the location of the actual error may have been lost.

- After an error has occurred, the parser must pick a likely place to resume the parse. A parser should always try to parse as much of the code as possible, in order to find as many real errors as possible during a single translation.
Error Recovery in Parsers (continued)

- A parser should try to avoid the error cascade problem, in which one error generates a lengthy sequence of spurious error messages.

- A parser must avoid infinite loops on errors, in which an unending cascade of error messages is generated without consuming any input.
“Panic Mode” in recursive-descent

- Extra parameter consisting of a set of *synchronizing tokens*.
- As parsing proceeds, tokens that may function as synchronizing tokens are added to the synchronizing set as each call occurs.
- If an error is encountered, the parser scans ahead, throwing away tokens until one of the synchronizing set of tokens is seen in the input, whence parsing is resumed.
Example (in pseudocode)

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { EOF } ) do
    getToken ;
  end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset ∪ followset ) ;
  end if ;
end;
```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = – do
      match (token) ;
      term ( synchset ) ;
    end while ;
  checkinput ( synchset, { (, number } ) ;
  end if;
end exp ;
Example (in pseudocode, concl.)

procedure factor ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    case token of
      ( : match( ( ) ; exp ( { } ) ) ; match( ) ) ;
      number : match(number) ;
    else error ;
    end case ;
  checkinput ( synchset, { (, number } ) ;
  end if ;
end factor ;