1 Top-Down Parsing

- Top-down vs. bottom-up
  - less powerful
  - more complicated to generate intermediate code or perform semantic routines
  - easier to implement, especially recursive descent

- Top-down parser
  - builds the parse tree from the root down
  - follows leftmost derivation; herefore, it is called $LL(k)$
  - we always expand the topmost symbol on the stack (leftmost in derivation)

- Top-down procedure ($k=1$ lookahead)
  - utilize stack memory
  - start with pushing initial nonterminal $S$
  - at any moment
    - if a terminal is on the top of the stack
      - if it matches the incoming token, the terminal is popped and the token is consumed
      - error otherwise
    - if a nonterminal is on the top of the stack
      - we need to replace it by one of its productions
      - must predict the correct one in a predictive parser (if more than one alternative)
      - errors possible in a predictive parser if none predicted
  - successful termination
    - empty stack and $\text{EOF}\_tk$ incoming
    - empty stack is an error otherwise

- Some other properties
  - actual program tokens are never on the stack
  - the stack contains predicted tokens expected on input
  - we must make correct predictions for efficiently solving alternatives
Example 1.1 Use the unambiguous expression grammar to top-down parse \texttt{id+id*id}.

Problems to handle in top-down parsing

- left recursive productions (direct or indirect)
  - infinite stack growth
  - can always be handled
- non-deterministic productions (more than one production for a nonterminal)
  - may be handled, or not, by left-factorization
  - verified by \textit{First} and \textit{Follow} sets

2 Left Recursion

Top-down parsers cannot handle left-recursion

- any direct left-recursion can be removed with equivalent grammar
- indirect left-recursions can be replaced by direct, which subsequently can be removed

Removing direct left recursion:

- separate all left recursive from the other productions for each nonterminal
- \( A \rightarrow A\alpha \mid A\beta \mid ... \)
  
  \[ A \rightarrow \gamma_1 \mid \gamma_2 \mid ... \]

- introduce a new nonterminal \( A' \)

- change nonrecursive productions to

- \( A \rightarrow \gamma_1 A' \mid \gamma_2 A' \mid ... \)

- replace recursive productions by

\[ A' \rightarrow \varepsilon \mid \alpha A' \mid \beta A' \mid ... \]

Example 2.1 Remove left recursion from \( R = \{ E \rightarrow E + T \mid T \} \)

\[ R = \{ E \rightarrow TA' , A' \rightarrow T A' \mid \varepsilon \} \]
Removing all left recursions (direct and indirect):

- Order all nonterminals on a list.
- Sequence through the list; For each nonterminal B
  - for all productions $B \rightarrow A\beta$, where $A$ precedes $B$ on the list
  - suppose all productions for $A$ are $A \rightarrow \alpha_1 | \alpha_2 | ...$
  - replace them by $B \rightarrow \alpha_1\beta | \alpha_2\beta | ...$
  - when finished, remove all immediate left recursions for $B$

3 Non-determinism

- Often grammar is nondeterministic
  - more than one choice of a production per non-terminal
- Backtracking implementation is infeasible
  - we use lookahead token to make predictions
  - if $k$ tokens are needed to look ahead, then grammar is LL($k$)
  - left-factorization reduces $k$
  - there are LL($k>1$) grammars that are not LL($k-1$)

Example 3.1 if then [else] is an example of not-LL(1) construct, but it may be solved in LL(1) by ordering productions.

- For predictions, use
  - left factorization to alter grammar (reduce $k$)
  - FIRST and FOLLOW sets to verify and construct actual predictions
3.1 **Left factorization**

- Combines alternative productions starting with the same prefixes
  - this delays decisions about predictions until new tokens are seen
  - this is a form of extending the lookahead by utilizing the stack
  - bottom-up parsers extend this idea even further

---

**Example 3.2** \( R=\{S \rightarrow ee \mid bAc \mid bAe, A \rightarrow d \mid cA\} \) has problems with \( w=bcde \). Change to \( R=\{S \rightarrow ee \mid bAQ, Q \rightarrow c \mid e, A \rightarrow d \mid cA\} \).

---

- FIRST and FOLLOW sets are means for verifying LL(\( k \)) and constructing actual predictions
  - they are sets of tokens that may come on the top of the stack (consume upcoming token)
  - parsers utilizing them are called *predictive parsers*

**FIRST(\( \alpha \))** algorithm:

- If \( \alpha \) is a single character or \( \epsilon \)
  - if \( \alpha \)=terminal \( y \) then \( \text{FIRST}(\alpha)=\{y\} \)
  - if \( \alpha=\epsilon \) then \( \text{FIRST}(\alpha)=\{\epsilon\} \)
  - if \( \alpha \) is nonterminal and \( \alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \) then \( \text{FIRST}(\alpha)=\cup \text{FIRST}(\beta_i) \)

- \( \alpha=X_1X_2\ldots X_n \)
  - set \( \text{FIRST}(\alpha)=\{\} \)
  - for \( j=1..n \) include \( \text{FIRST}(X_j) \) in \( \text{FIRST}(\alpha) \), but break when \( X_j \) is not nullable
  - if \( X_n \) was reached and is nullable then include \( \epsilon \) in \( \text{FIRST}(\alpha) \)

---

**Example 3.3** \( R=\{S \rightarrow Ab \mid Bc, A \rightarrow Df \mid CA, B \rightarrow gA \mid e, C \rightarrow dC \mid c, D \rightarrow h \mid i\} \)

\( \text{FIRST}(Ab)=\{h,i,c,d\}, \text{FIRST}(Bc)=\{e,g\} \).
Section - Top-Down Parsings: 3  Non-determinism

- FOLLOW(A ∈ N):
  - Note that FOLLOW(A) never contains ε
  - Note that we compute FOLLOW of left hand side if there is empty production alternative
  - If A=S then put end marker (e.g., EOF token) into FOLLOW(A) and continue
  - Find all productions with A on rhs: Q → αAβ
    - if β begins with a terminal q then q is in FOLLOW(A)
    - if β begins with a nonterminal then FOLLOW(A) includes FIRST(β) - {ε}
    - if β=ε or when β is nullable then include FOLLOW(Q) in FOLLOW(A)

- Grammar is one-lookahead top-down predictive, LL(1), when
  - or every pair of productions with the same lhs such as X → α | β
    - First(α)-ε and First(β)-ε are disjoint
    - if α is nullable (i.e., α=ε or ε∈FIRST(α)) then First(β) and Follow(X) are disjoint
    - same for β

- LL(k>1) parsers are generally infeasible due to size (program or table)

Example 3.4  Check if the unambiguous expression grammar is LL(1). Use $ as end-marker.
1) Remove left-recursion (apply)
2) Apply left-factorization (none)
Resulting rules={E → TQ, Q → +TQ|-TQ|ε, T → FR, R → *FR|/FR|ε, F → (E)|id}
E is unambiguous
Q has alternative productions:
  First(+TQ)={+}
  First(-TQ)={-}
  Follow(Q)=Follow(E)={$,} )
T is unambiguous
R has alternatives
  First(*FR)={*}
  First(/FR)={/}
  Follow(R)={+,-,$}
F has alternatives
  First((E))={ (}
  First(id)={id}
Grammar is LL(1).
How to use predict?

- suppose grammar is LL(1)
- suppose that $X$ is the next top of the stack symbol
- suppose that $tk$ is the next input token (one lookahead)
- suppose $X$ is nondeterministic with $X \rightarrow \alpha \mid \beta$
- compute $\text{First}(\alpha)$ and $\text{First}(\beta)$
  - if $tk$ is in neither $\text{First}(\alpha)$ nor $\text{First}(\beta)$ then error
  - if $\epsilon \notin \text{First}(\alpha)$ then predict $X \rightarrow \alpha$ when $tk \in \text{First}(\alpha)$
  - if $\epsilon \in \text{First}(\alpha)$ then predict $X \rightarrow \alpha$ when $tk \in \text{First}(\alpha) \cup \text{Follow}(X)$

4 Process Memory, Stack, Activation Records

Each process operates in its own (virtual) process space
- size depending the addressing space and user’s quota
- in older OS heap space could have been common between processes, resulting in one process bring down other processes or even the OS
- a process doesn’t have direct access outside of the process space
- elements
  - code
    - main, functions
  - persistent space
    - global data, local persistent data
  - stack
    - function call management with Activation Records (AR)
  - heap
    - dynamic memory, controlled by heap manager
      - under direct control of the program (C/C++)
      - garbage collection (Java)
4.1 Stack and ARs

- Stack is accessed indirectly (HLL) to manage
  - function calls
  - local scopes
- Compiler generates one AR per function
  - AR is a memory template specifying the relative location of the AR elements
    - automatic data
    - parameters and returning data
    - address of the next instruction
    - Static Link
      - used for accessing data in enclosed scopes
      - not needed in languages w/o scoped functions
    - Dynamic Link
      - pointing to the previous AR
- actual activation records are allocated on the stack for each function call
  - multiple allocations for recursive calls
  - TOS is always the AR for the currently active function

- Heap and Stack may compete for the same storage
Example 4.1  Example of ARs and runtime stack. Assume main calls $f(2)$. Details of the AR for main are not shown.

```c
int global;

void f(int x) {
    g(x); // intr 1
}
void g(int y) {
    int g1;
    if (y>1) g(y-1);
    else return; // instr 3
}

AR(f)
0
x
Ret
DL

AR(g)
0
y
g1
Ret
DL

```

```
main, f, g

global

heap

y=1
g1
Ret = 3
DL

y=2
g1
Ret = 3
DL

x
Ret = 1
DL

AR(main)
```

Programming Language Processors © 1999 Cezary Z. Janikow
Page 8
5 Recursive Descent Parsing

- Top-down parsers can be implemented as recursive descent or table-driven.
- Recursive descent parsers utilize the machine stack to keep track of parse tree expansion. This is very convenient, but may be inefficient due to function calls.
- Recursive descent parser performs two actions:
  - if the next symbol (leftmost derivation) in a production is a terminal and it matches the next token in the sentence then the token is consumed.
  - if the next symbol in the production is a nonterminal then a routine is called to recognize that nonterminal.
- Initially, the starting nonterminal is used. Recognition succeeds only when the sentence is exhausted and there is noting more in productions.

**Example 5.1** Try to write a recursive descent parser for \( R = \{ S \rightarrow bA \mid c, A \rightarrow dSd \mid e \}. \)

Assumptions: Each function gets and returns a fresh token.

```
program
    token=getToken();
    S(token);
    if (token==EOF) OK
    else ERROR;
```