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 m
  - 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
    - only on empty stack and `EOFtk` incoming
- 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 `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 *First* and *Follow* sets (must be pairwise disjoint for the same nonterminal)

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={$E \rightarrow TA'$, $A' \rightarrow +TA' \mid \varepsilon$}

- Removing all left recursions (direct and indirect):
  - Order all nonterminals (new added noterminals go in the rear)
  - 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 $A \rightarrow \alpha_1 \mid \alpha_2 \mid ...$
- replace them by $B \rightarrow \alpha_1\beta \mid \alpha_2\beta \mid ...$
- 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) - as said before, left-factorization may or may not help

Example 3.1 if then [else] is an example of not-LL(1) construct and cannot be reduced to LL(1), but it may be solved in LL(1)-parser using other technques such as by ordering productions.

- For predictions, use
  - left factorization to alter grammar (reduce *k*) if needed
  - *FIRST* and *FOLLOW* sets to verify and construct actual predictions
    - given nondeterministic production
      $N \text{ -> } \alpha \mid \beta$
      compute FIRST of non-empty right hand sides
      compuet FOLLOW(N) is also -> ε
    - all sets must be disjoint pairwise (not account for empty symbol) for the production to be LL(1)
    - Th entire grammar is LL(k) where k is the max (LL?) over each production

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→ee | bAc | bAe, A→d | cA} has problems with w=bcde. Change to R={S→ee | bAQ, Q→c | e, A→d | 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(α) algorithm:
  Note that we compute FIRST of right hand sides of non-deterministic productions and not individual nonterminals unless needed in the algorithm
  - If α is a single element (terminal/token or nonterminal) or ε
    - if α=terminal y then FIRST(α)={y}
    - if $\alpha=\varepsilon$ then FIRST(α)={ε}
    - if α is nonterminal and $\alpha \to \beta_1 \mid \beta_2 \mid \ldots$ then FIRST(α)=$\cup$ FIRST(β_i)
  - $\alpha=X_1X_2...X_n$
    - set FIRST(α)={}
    - for j=1..n include FIRST(X_j) in FIRST(α), but STOP when X_j is not *nullabl*e
      - X is nullable if X -> ε, directly or indirectly
      - terminal is of course never nullable
    - if X_n was reached and is also nullable then include ε in FIRST(α)

Example 3.3 R={S→Ab | Bc, A→Df | CA, B→gA | e, C→dC | c, D→h | i}
FIRST(Ab)={h,i,c,d}, FIRST(Bc)={e,g}.

- FOLLOW($A \in N$) :
  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: $Q \rightarrow \alpha A \beta$
  - 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 $\beta=\varepsilon$ 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 $X \rightarrow \alpha \mid \beta$
    - First(α)-ε and First(β)-ε are disjoint
    - if α is nullable (*i.e.*, $\alpha=\varepsilon$ or $\varepsilon \in$ 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 apparent)
Resulting rules={
  $E \rightarrow TQ$
  $Q \rightarrow +TQ|-TQ|\varepsilon$
  $T \rightarrow FR$
  $R \rightarrow *FR|/FR|\varepsilon$
  $F \rightarrow (E)|id$
}

E is unambiguous	LL(0)
Q has alternative productions:	LL(1)
First(+TQ)={+}	
First(-TQ)={-}	
Follow(Q)=Follow(E)={$,)}	
T is unambiguous	LL(0)
R has alternatives	LL(1)
First{*FR)={*}	
First(/FR)={/}	
Follow(R)={+,-,),$}	
F has alternatives	LL(1)
First((E))={(}	
First(id)={id}	

Grammar is LL(1) (max value among all non-terminals).

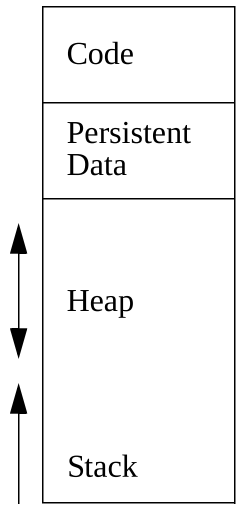
- How to use predict?

- suppose grammar is LL(0) or LL(1)
  - meaning all sets for the same nonterminal are pairwise disjoint
- suppose that is the next top of the stack symbol
- suppose that `tk` is the next input token (one lookahead)
- suppose `X` is determistic with $X \to \alpha$
  - use the production, even if empty, no predictions needed
- suppose `X` is nondeterministic with $X \to \alpha \mid \beta$
  - compute `First`(α) and `First`(β)
  - if `tk` is in neither `First`(α) nor `First`(β) then error
  - if $\varepsilon \notin$ `First`(α) then predict $X \to \alpha$ when `tk`$\in$`First`(α)
  - if $\varepsilon \in$ `First`(α) then predict $X \to \alpha$ when `tk`$\in$`First`(α)$\cup$`Follow(X)`
  - same on β
- suppose `X` is nondeterministic and includes the empty production as well
  - as above, except that predict the empty production when `tk`$\in$`Follow(X)`
  - even better - predict the empty production when no other production can be predicted

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)

Figure 4.1 Process space (Heap and Stack can be reversed).



- Heap and Stack may compete for the same storage

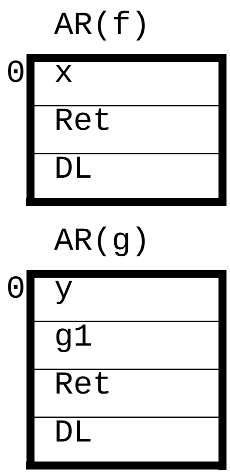
4.1 Stack and ARs

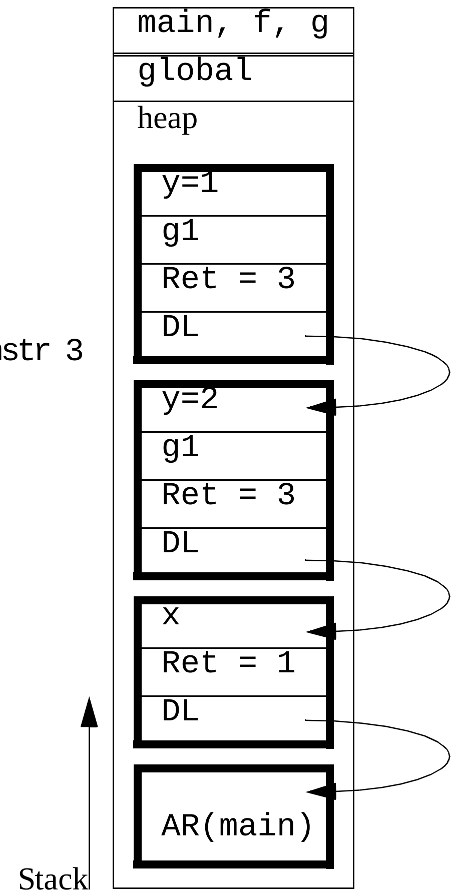
- Stack is accessed indirectly (HLL) to manage
  - function calls
  - local scopes
- Compiler generates one AR per function
  - AR is a memory template specifying the relative location of the AR elements
    - automatic data
    - parameters and returning data
    - address of the next instruction
    - 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 recursvie calls
    - TOS is always the AR for the currently active function

Example 4.1 Example of ARs and runtime stack. Assume main calls `f(2)`. Details of the AR for main are not shown.

```
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
}
```





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 less inefficient in larger compilers due to function calls
- Recursive descent parser processes production right hand sides following leftmost derivation (left to right), one right hand side at a time, and performs two actions:
  - if the next symbol is a terminal and it matches the next token in the sentence then the token is matched against the token from the scanner
    - consumed if matching (get the next token)
    - error if not matching
  - if the next symbol is a nonterminal then a function is called to process that nonterminal
- Initially, the starting nonterminal is used (that is, the function for the starting nontermonal is called)
- Recognition succeeds only when the input is exhausted (EOF token reached) and the inital call (see above) returns
    - this needs to be checked in the function that made the initial call to the initial nonterminal. This function can be the main function or some additional parser() function
- Each function is implemented to perform
  - Prediction if needed (if more than one production on a nonterminal)
  - The two basic actions while moving left-to-right in on the symbols of the predicted right hand side
- First modify if needed and validate grammar to be LL(1) by computing needed First and Follow sets
- Useful assumptions to consider
  - each function is called with unconsumed token, and returns with unconsumed token
  - each nonterminal has a corresponding single function named after the nonterminal

5.1 Parsing

- Parsing is accomplished as detailed above
- With only parsing, the only possible outcome is

  - OK - when the scessfull termination encountered (see above)
  - Error - error message and exit, no recovery
- If grammar is validated as LL(1), you can predict the empty production (if applicable) when no other production is predicted (no need to check the Follow set)
- Suggested to use void functions and use all explicit returns only

Example 5.1 Try to write a recursive descent parser for
R={ $S \rightarrow bA \mid c$
$A \rightarrow dSa \mid \varepsilon$
}.
First, validate LL(1)
S: First(bA) = {b}, First(c) = {c} thus LL(1)
A: First(dSa) = {d}, Follow(A) = {a, $} thus LL(1)
Thus the grammar is LL(1).
Assume `tk` is a token storage available and modifiable in all functions, and assume `scanner()` reurns the next token

```
void parser(){
   tk=scanner();
   S();
   if (tk.ID == EOFtk)
      printf("Parse OK");
   else error(); // error message detail not shown, exit, no recovery
}

void S()
   if (tk.ID == b) {         // predicts S→bA since b∈First(bAa
      tk=scanner();          // processing b, consume matching tokens
      A();                   // processing A
      return;
   }
   else if (tk.ID == c) {    // predict S→c
      tk=scanner();          // consume c
      return;                // explicit return
   }
   else error();
}

void A()
   if (tk.ID == d) {         // predicts A→dSa
      tk=scanner();          // processing d
      S();                   // processing S
      if (tk.ID == a) {      // processing a
         tk=scanner();
         return;
      }
      else error();
   }
   else                      // predicts A→ε
      return;                // explicit return
                             // or could check tk ∈ Follow(A)
```

5.2 Tree Generation

- The parser above can be easily modified to generate parse tree, with the following changes
  - every function generates zero or one node and returns pointer to what was generated
  - every function stores or disposes tokens that are consumed
    - structural tokens are disposed, semantics tokens are stored
- Useful assumptions and suggestions to modify recursive descent parser to generate the parse tree
  - every function creating a node will label the node with the name of the nonterminal function
  - every function making calls to other nonterminal function(s) will collect returned pointers and attach to its node
  - every function will return its node
  - every function will store tokens carrying any semantics information (ID, number, operator)
  - the maximum number of children is the maximum number of nonterminal in any production
  - the maximum number of tokens to be stored in a node is the maximum number of semantics tokens in any production

---

Example 5.2 Modify the previous code to generate parse tree. Suppose b and d tokens need to be retained while c does not.
Max one child needed per node since at most one nonterminal on the right hand side of any production.
Max two tokens need to be stored in a node as one production use two semantics tokens.
Assume `node_t` structure with `label`, `token1`, `token2`, and `child`.
Assume `getNode(label)` allocates `node_t` node and labels it.

```
node_t parser(){
   node_t *root;
   tk=scanner();
   root = S();
   if (tk.ID == EOFtk)
      printf("Parse OK");
   else error(); // error message detail not shown, exit, no recovery
   return root;
}
```

```
node_t* S()
   node_t *p = getNode(S);    // label the node with S
   if (tk.ID == b) {
      p->token1 = tk;         // store b in the tree
      tk=scanner();
      p->child = A();
      return p;
   }
   else if (tk.ID == c) {
      tk=scanner();           // c not needed to be stored
      return p;               // could also return NULL since
   }                          // the node is empty in this case
   else error();
}

node_t* A()
   if (tk.ID == d) {
      token_t p = getNode(A); // label the new node with A
      p->token1 = tk;
      tk=scanner();
      p=>child = S();
      if (tk.ID == a) {
         p->token2 = tk;
         tk=scanner();
         return p;
      }
      else error();
   }
   else
      return NULL;            // empty production so no node generated
```