Recursive Descent Parser

Grammar LL(1)

- Grammar does NOT have left recursions
- Every nonterminal with multiple right hand side (RHS) productions has disjoint First sets on those productions
    - In addition, every nonterminal with the empty production has the Follow set disjoint with the First sets on the RHS productions
        - For both direct and indirect empty production

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 as no explicit stack manipulations needed
        - We utilize automatic operations on the system stack
            - function call is activation record (AR) push
            - function return is pop
    - Less efficient due to function calls
        - Good enough for compiling smaller programs
- Errors can be accumulated and repairs attempted but the simplest case is to display a detailed error message and exit upon error

Implementation

- **BNF is available (assuming LL(1))**
- **There is one auxiliary function**
    - Called first to start parsing
    - Gets the first token from the scanner to be available for inspection (lookeahed)
    - Calls the function for the starting BNF nonterminal
        - When that function returns, the auxiliary function checks that the next token is EOFtk and declares success if yes or error otherwise
- **Every nonterminal is implemented as a function**
    - Every function is implemented as follow
        - If there is a single production, process the elements left to right
            - If the element is a token
                - If it matches the token from the scanner
                    - the scanner token gets **consumed**
                    - get new token from the scanner – **unconsumed** until matched

            - continue to the next element
        - Else, an error
    - If it is a nonterminal, call its function
    - At the end of the production, return
- Else, predict one production based on the current token from the scanner
    - Select for processing the production whose First set contains the token
    - Else if the token doesn't match any First sets
        - if there is the empty production, return from the function
        - else an error

Implementation Notes/Suggestions

- Modify grammar as needed until LL(1)
- Implement the extra auxiliary function
- Implement one function per nonterminal as explained above, named same as the nonterminal
- Each function is `void` to allow building the tree later
- Keep the scanner token unconsumed
    - Get new token whenever it gets consumed
    - Each function is called with unconsumed token and returns with unconsumed token
- Use only explicit returns only to allow building the tree later
    - Implicit function return is when function execution gets to the closing } for the function

Example

Build recursive descent parser for this BNF (shown as LL(1)). Lower cases are tokens.

S -> bA | c	FIRST(bA)={b} FIRST(c)={c} thus	k=1
A -> dSa | ε	FIRST(dSa)={d} FOLLOW(A)={a EOFtk}	k=1

Assume `tk` variable is available and modifiable in all functions, and `scanner()` returns the next token.

```
void parser(){              // auxiliary, always the same
       tk=scanner();        // get first token
       S();                 // call the first nonterminal
       if (tk == EOFtk)
              return ok;    // continue, parse was ok
       else error("tk received, EOFTk expected");     // will exit, no recovery
}
void S() {                  // k=1 thus predictions needed compared against First sets
       if (tk == b) {       // predicts S->bA since First(bA)={b}
              tk=scanner();        // tokens match, consume, get new one
              A();                 // processing A
              return;              // done, explicit return
       }
       else if (tk == c) { // predict S->c since First(c)={c}
              tk=scanner();        // tokens match, consume, get new one
```

```
            return;             // done, explicit return
        }
        else error("…");
}
void A() {
        if (tk == d) {          // predicts A->dSa since First(dSa)={d}
            tk=scanner();       // consume matching d, get new token
            S();                // processing S
            if (tk == a) {      // processing a
                tk=scanner();
                return;
            }
            else error("…");    // processing this production requires token a now
        }
        else                    // predicts A->ε, could check the Follow set
            return;             // explicit return on empty production
}
```

Tree Generation in Recursive-Descent Parser

- The parser above can be easily extended to generate parse tree, with the following changes
  - Every function generates a node (function that returns on the empty production can return null) and returns pointer to what was generated
  - Every function stores or disposes program tokens (from the scanner) that are consumed
    - structural tokens are disposed: {}, (), delimiters, *etc*.
    - semantics tokens are stored in the node: IDTk, NumberTk, OperatorTk, *etc*.

Useful assumptions and suggestions to modify recursive descent parser to generate the parse tree

- Every node will contain
  - A label identifying the function that created the label (and thus nonterminal)
    - Use enumeration, or string the same as function name (same as nonterminal)
  - Potential token(s)
    - Only sementic tokens, those that cannot be removed from the parse tree, must be stored
  - Potential children in the tree
- Every function making calls to other nonterminal function(s) will collect returned pointers and attach to its node children pointers, left to right
- Every function will return its node (or possibly null)
- Every function will store processed semantics tokens
- The maximum number of children per node is the maximum number of nonterminal in any production

- The maximum number of tokens per node is the maximum number of semantics token in any production

Example

Modify the previous code to generate the parse tree. Suppose **b** and **d** are semantics tokens (need to be retained) while **a** and and **c** are structural tokens (can be thrown away).

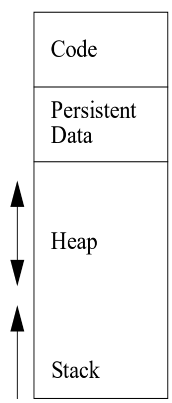
- Max one child needed per node
  - The is max 1 nonterminal on the right hand sides per production
- Max one token needed to be stored in a node
  - **dSa** has 2 tokens (maximum across all productions) but **a** can be thrown away
- Assume `node_t` structure with label, token, and child.
- Assume `getNode(label)` allocates `node_t` node and labels it according to the parameter.

```
node_t* parser(){
        node_t* root;
        tk=scanner();
        root = S();                 // Auxilary function does not allocate node
                                    // but returns the node of S()
        if (tk.ID == EOFtk)
                return root;        // parse was ok
        else error("…");            // error message, exit, no recovery
}
node_t* S() {
        node_t* p = getNode(S);     // label the node same as function name
        if (tk.ID == b) {           // predicts S->bA since b ϵ First(bAa)
                p->token = tk;      // b needed to be stored
                tk=scanner();       // consume matching token, get new one
                p->child = A();     // processing A
                return p;
        }
        else if (tk.ID == c) {      // predict S->c
                tk=scanner();       // consume c, no need to store
                return p;           // explicit return
        }
        else error("…");
}
        // note the getNode() call could be the first statement
        // resulting in allocating  empty node on the empty production
node_t* A() {
        if (tk.ID == d) {                   // predicts A->dSa
                token_t* p= getNode(A);     // could be before if() - node always created
                p->token = tk;              // d needed to be stored
```

```
            tk=scanner();               // get next token
            p->child = S();             // processing S
            if (tk.ID == a) {           // processing a
                  tk=scanner();
            }
            else error();
            return p;
      }
      else                              // predicts A->ε
            return NULL;
}
```

Process Space and Stack

- 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
- Process space parts
  - Code
    - main, functions, linked library functions in static linking
    - each function is always there just once, regardless of whether will be called and how many times
  - Persistent space
    - global data (exposed and static), local persistent data
  - Stack
    - function call management with Activation Records (AR) including local data and parameters
  - Heap
    - dynamic memory, controlled by heap manager and managed by programmer (C/C++) or garbage collection (Java)



System Stack and Activation Records

- System stack is accessed indirectly (in HLL) to manage
  - Function calls
  - Memory for local scopes: local variables and parameters
- 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 or blocks
- Dynamic Link
  - pointing to the previous AR

- Actual activation records are allocated on the stack for each function call
  - Multiple allocations for recursive calls
  - A function never called will never have its activation record on the stack
  - TOS is always the activation record for the currently active function

Example

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
}
```