Recursive Descent Parser

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
- Every nonterminal has a function, which does prediction if needed and then applies one RHS
- Errors can be accumulated but the simplest case error displays message and terminates the parser
- Recursive descent parser starts with some auxiliary function called, which then
    - Gets the first token from the scanner
    - Calls the function for the starting nonterminal
        - When this function returns, the auxiliary function checks that the next token is EOFtk and declares success or error
- Every function works the same way
    - If there is a single RHS, process it
    - Otherwise, predict one RHS based on the current token from the scanner
        - Select the RHS whose FIRST set contains the token
        - Otherwise, if there is the empty production, return from the function
        - Otherwise an error
- Processing the predicted RHS involves:
    - Process the symbols in that RHS left to right, one at a time
        - If it is a token
            - If it matches the token from the scanner, get new token from the scanner and continue next symbol
            - Else an error
        - If it is a nonterminal, call its function
        - At the end of RHS, return
- Implementation notes
    - Implement the extra auxiliary function to check the termination condition
    - Implement one function per nonterminal
    - Each function is called with unconsumed token, and returns with unconsumed token
    - Each function is `void` to allow building the tree later
    - Do not use implicit returns

Example

Build recursive descent parser for

S -> bA | c FIRST(bA)={b} FIRST(c)={c} thus LL(1)

A -> dSa | ε FIRST(dSa)={d} FOLLOW(A)={a EOFtk} thus LL(1)

Assume tk is a token storage available and modifiable in all functions, and assume scanner() returns the next token.

```
void parser(){
	tk=scanner();
	S();
	if (tk.ID == EOFtk)
				// continue, parse was ok
	else error();	// error message, exit, no recovery
	return;
}
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□□¢
		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
```

Tree Generation in recursive-Descent Parser

- The parser above can be easily modified to generate parse tree, with the following changes
  - every function generates zero or one node and returns null or pointer to what was generated
  - every function stores or disposes tokens that are consumed
    - structural tokens are disposed
    - semantics tokens are stored in the node

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

- Every node will contain
  - Label identifying the function that created the label (and thus nonterminal)
  - Potential token(s)
  - Potential chidren
- Every function making calls to other nonterminal function(s) will collect returned pointers and attach to its node children pointers
- Every function will return its node or null
- Every function will store processed tokens carrying any semantics information (ID, number, operator)
- The maximum number of children is the maximum number of nonterminal in any production

Example

Modify the previous code to generate the parse tree. Suppose **b** and **d** tokens need to be retained while **a** and and **c** do 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.

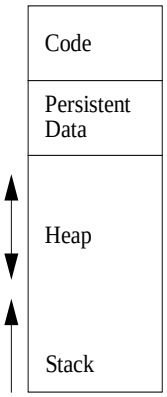
Assume `node_t` structure with label, token1, token2, and child.

Assume `getNode(label)` allocates `node_t` node and labels it.

```
node_t* parser(){
	node_t* root;
	tk=scanner();
	root = S();
	if (tk.ID == EOFtk)
				// continue, parse was ok
	else error();	// error message, exit, no recovery
	return root;
}
node_t* S() {
	node_t* p = getNode(S);
	if (tk.ID == b) {   // predicts S->bA since b∈First(bAa)
		p->token1 = tk;	// b needed to be stored
		tk=scanner();	// processing b, consume matching tokens
		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();
}
node_t* A() {
	if (tk.ID == d) {		// predicts A->dSa
		token_t* = genNode(A);	// now we know we need node
		p->token = tk;		// d needed to be stored
		tk=scanner();		// processing d
		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
  - Persistent space
    - global data, 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 tack is accessed indirectly (HLL) to manage
  - Function calls
  - Memory for 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 or blocks
- Dynamic Link
  - pointing to the previous AR

o 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

Example

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