**Interpreting LL Parsing Table**

This is the top-down parser:

[var](http://scala-lang.org) stack : Stack[GrammarSymbol]

stack.push(EOF);

stack.push(StartNonterminal);

lex = [new](http://scala-lang.org) Lexer(inputFile)

[while](http://scala-lang.org) ([true](http://scala-lang.org)) {

\* X = stack.pop

t = lex.curent

[if](http://scala-lang.org) isTerminal(X)

[if](http://scala-lang.org) (t==X)

[if](http://scala-lang.org) (X==EOF) [return](http://scala-lang.org) success

[else](http://scala-lang.org) lex.next // eat token t

[else](http://scala-lang.org)

parseError("Expected " + X)

[else](http://scala-lang.org) // non-terminal

cs = choice(X)(t)

cs [match](http://scala-lang.org) {

[case](http://scala-lang.org) {i} => // exactly one choice

rhs = p(X,i) // choose correct right-hand side

\* stack.push(reverse(rhs))

[case](http://scala-lang.org) {} => parseError("Parser expected an element of " + unionOfAll(choice(X)))

[case](http://scala-lang.org) \_ => crash("wrong parse table, not LL(1)")

}

}

The lines marked with \* give us the essence of this parser: it pops non-terminals from stack and replaces them with the right-hand side of a production rule.

When we write recursive descent procedures by hand, the stack is implicit in the use of recursive procedures.

Note: the program above corresponds to a deterministic push-down automaton that parses the LL(1) grammar

* non-deterministic push down automata correspond to all grammars
* determinization of push down automata in general is not possible, non-deterministic ones are more expressive