

Top-down parsing

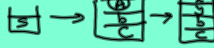
stack for the parser:

Given grammar G
start symbol S

S expands:
replace with
body of
production
which is
the next token
on top of stack

no advance
of the
current
scanner

$S \rightarrow A b C$



$A \rightarrow x y$

match now: top of stack
is a terminal which must
match the next token in
the input ("lookahead")
if not, syntax error

~~Initially, stack is empty~~
while stack not empty:

if top is nonterminal,
expand by some production
headed by that nonterminal
(which production
lookahead tells you)

else if top is token,
then match
(also syntax error if
no more input)

end-while

if not at end of input,
also syntax error

Between actions, take a
snapshot of

input string and current (top down)

Follows leftmost derivation of input
(forward)

S
 $A b C$
 $x y b C$
 $x y b C$
 $x y b C$
 $x y b C$
 $x y b$

LL parsing Left-to-right
reducing
Leftmost derivation
(forward)

LL(1) - LL parsing with
one token of lookahead

Top-down parsing

advantage: much easier to
implement

disadvantage: only certain
grammars are
suitable

$E \rightarrow E + T \mid E - T \mid T$

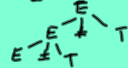
$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow c \mid v \mid (E)$

$2 + 4 * 5$

E

$E \rightarrow T \pm T \pm T \pm T \pm T$

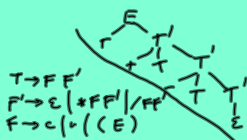


Equivalent LL(1) grammar:

$E \rightarrow T T'$

$T' \rightarrow \epsilon \mid + T T' \mid$

$- T T'$



```

<expr> ::= <term> <moreterms>
<moreterms> ::= '+' <term> <moreterms>
              | '-' <term> <moreterms>
              | /* empty */
<term> ::= <factor> <morefactors>
<morefactors> ::= --
<factor> ::= const | '(' <expr> ')'

Recursive descent, predictive parser.
For each nonterminal
  write a function (same name)
  whose job is to read
  enough input that is yielded
  by the next terminal.
  and return app. val.

```

```

enum { LEX_CONST, LEX_VAL,
       LEX_EOF };

int lookahead; // type of next
               // token to be
               // read
int val; // attribute val of
         // the lookahead
void gettoken(); // advance lookahead
               // like yylex()

int expr()
// read an expression from stdin,
// return its value
// Precondition: lookahead is
// first token in the expr
// Postcondition: lookahead is
// token immediately following
// the expression
// <expr> ::= <term> <moreterms>
{
    int val;
    val = term();
    val += moreterms();
    return val;
}

int moreterms()
// <moreterms> ::= '+' <term> <moreterms>
// | '-' <term> <moreterms>
{
    int val = 0;
    if (lookahead == '+') {
        gettoken();
        val = term();
        val += moreterms();
    }
    else if (lookahead == '-') {
        gettoken();
        val = -term();
        val += moreterms();
    }
    return val;
}

int term()
{
    int val;
    val = factor();
    val = morefactors(val);
    return val;
}

int morefactors(int val)
{
    if (lookahead == '*') {
        gettoken();
        val *= factor();
        val = morefactors(val);
    }
    else if (lookahead == '/') {
        gettoken();
        val /= factor();
        val = morefactors(val);
    }
    return val;
}

int factor()
{
    int val;
    if (

```

```

int factor()
{
    int val;
    if (lookahead == LEX_CONST) {
        val = lval;
        gettoken();
        return val;
    }
    if (lookahead == LEX_VAR) {
        val = get_val(lval);
        gettoken();
        return val;
    }
    if (lookahead == '(') {
        gettoken();
        val = expr();
        if (lookahead != ')')
            error();
        else {
            // "right paren expected"
            gettoken();
            return val;
        }
    }
    error("left paren expected  
illegal first token  
of factor")
}

int main()
{
    int val;
    gettoken(); // initialize lookahead
    val = expr();
    if (lookahead != LEX_EOI)
        error("extra token after  
expr");
    printf("expr = %d\n", val);
}

```