CSE 3302 Notes 4: Lexical & Syntax Analysis

(Last updated 6/24/14 7:13 PM)

4.1. Connections of Scanning to Formal Languages (aside)

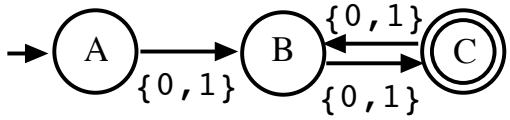
Deterministic FSAs

States (start, accepting)

Transitions

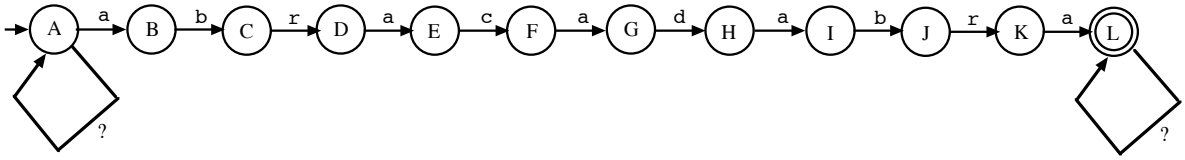
(*i*) ε ("free move") is prohibited on transitions
(*ii*) From a given state, two transitions cannot have the same label

Non-empty binary strings of even length:

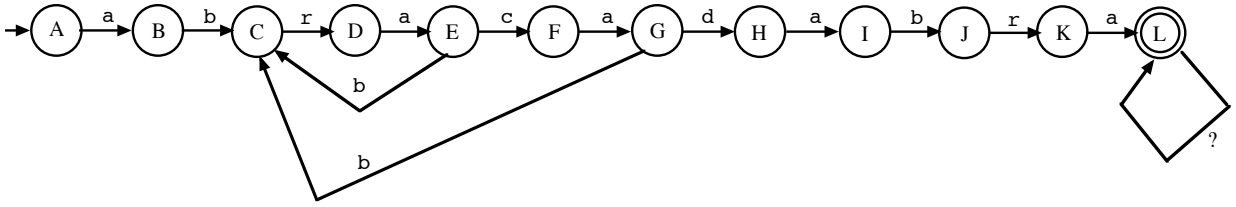


Non-deterministic FSAs (drop (*i*) and (*ii*) above)

Does input string contain `abracadabra`?



Aside: Deterministic counterpart



For each state, except L:

1. If a state has no outgoing transition shown for symbol `a`, then there is one to state B.
2. If a state has no outgoing transition for a symbol, then there is one to A.

Variation: Count the number of occurences (with overlap allowed) for `abracadabra`.

Languages described by REs, DFSAs, and NDFSAs?

$a^n b^n c^n$ example from Notes 03: http://ranger.uta.edu/~weems/NOTES3302/anbncn.html

Playing out-of-bounds (Power Lexer, http://ranger.uta.edu/~weems/NOTES3302/power.html)

```
collect=/^(.*)(.*)\1{2}\2{3}\1{4}\2{5}$/,result;

result=collect.exec(input);
if (result==null)
  myOutput.value = "search bonked";
else
  myOutput.value = result[1]+" "+result[2];
```

(Chapter 7, p.66 of *Good Parts* is useful, also http://proquestcombo.safaribooksonline.com.ezproxy.uta.edu/9781449393854/ch10_html)

4.2. Scanning Examples

"Cat Dog ++ +" in JavaScript (http://ranger.uta.edu/~weems/NOTES3302/catDog.html)

Lab 1 Fall 2012 in JavaScript - tokenizes expressions (http://ranger.uta.edu/~weems/NOTES3302/lab1fall12.html)

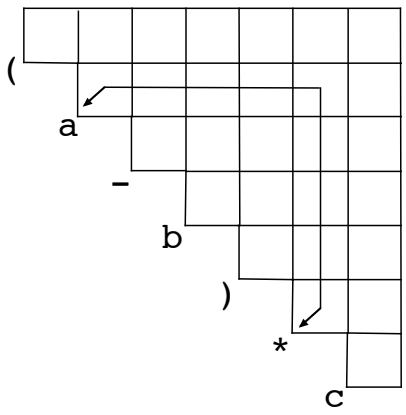
Sebesta 4.2 - hand coded lexer (length?)

Wirth: PL/0 - `getsym/getch`, Pascal-S - `insymbol/nextch`

4.3. Parsing

Brute Force Recursive (find handles and reduce) - $O(n^3)$ time

What non-terminal symbols have a derivation leading to each substring of the input?



expr → *term* | *expr add_op term*
term → *factor* | *term mult_op factor*
factor → `id` | `number` | `-` *factor* | `(` *expr* `)`
add_op → `+` | `-`
mult_op → `*` | `/`

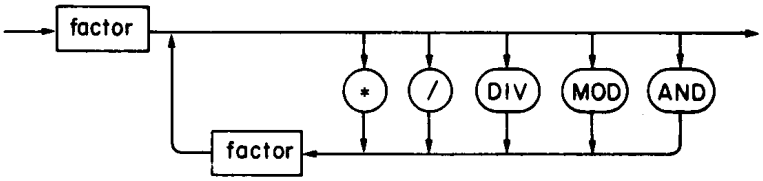
Form of grammar is usually restricted to simplify details and achieve time bound.

Similar to DP solution for optimal matrix multiplication

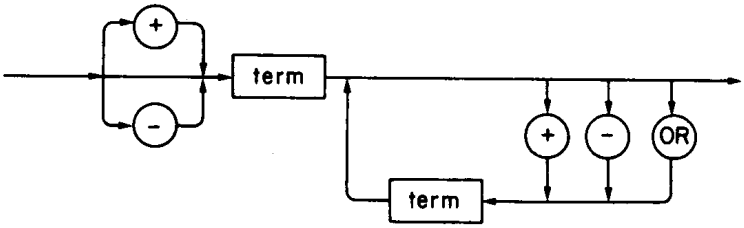
Recursive Descent (example of top-down)

"Pascal is a language that can be parsed with a lookahead of a single symbol. The compiler therefore uses the simple and efficient method of top-down parsing with one-symbol lookahead." (Pascal-S report)

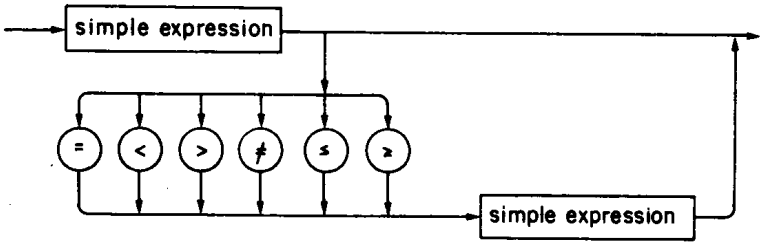
Term



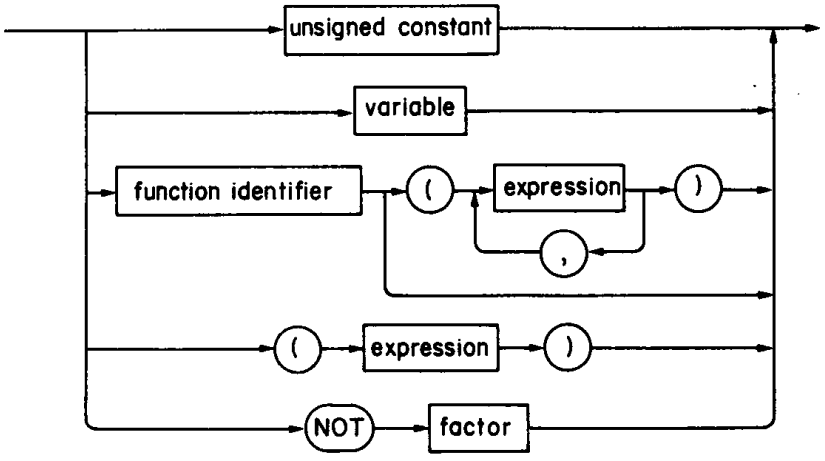
Simple expression



Expression



Factor



http://ranger.uta.edu/~weems/NOTES3302/LAB1SPR13/plzero.io.pas (see http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/adsprgs.pl0.pdf for syntax diagrams)

```
procedure test(s1,s2: symset; n: integer);
begin if not(sym in s1) then
        begin error(n); s1 := s1 + s2;
           while not(sym in s1) do getsym
        end
end {test};
…
      procedure expression(fsys: symset);
         var addop: symbol;
         procedure term(fsys: symset);
            var mulop: symbol;
            procedure factor(fsys: symset);
               var i: integer;
            begin test(facbegsys, fsys, 24);
               while sym in facbegsys do
               begin
                  if sym = ident then
                  begin i:= position(id);
                     if i = 0 then error(11) else
                     with table[i] do
                     case kind of
                        constant: gen(lit, 0, val);
                        varible: gen(lod, lev-level, adr);
                        proc: error(21) ;
                        instream: gen(rdi, 0, 0); {bpw}
                        outstream: error(21) {bpw}
                     end;
                     getsym
                  end else
                  if sym = number then
                  begin if num >  nmax then {BPW - shouldn't happen}
                           begin error(30); num := 0
                           end;
                     gen(lit, 0, num); getsym
                  end else
                  if sym = lparen then
                  begin getsym; expression([rparen]+fsys);
                     if sym = rparen then getsym else error(22)
                  end;
                  test(fsys, [lparen], 23)
               end
            end {factor};

         begin {term} factor(fsys+[times, slash]);
            while sym in [times, slash] do
             begin mulop:=sym;getsym;factor(fsys+[times,slash]);
              if mulop=times then gen(opr,0,4) else gen(opr,0,5)
             end
         end {term};
      begin {expression}
         if sym in [plus, minus] then
            begin addop := sym; getsym; term(fsys+[plus,minus]);
               if addop = minus then gen(opr, 0,1)
            end else term(fsys+[plus, minus]);
         while sym in [plus, minus] do
            begin addop := sym; getsym; term(fsys+[plus,minus]);
               if addop=plus then gen(opr,0,2) else gen(opr,0,3)
            end
      end {expression};
```

Advantages

Easy to use for small, "well-designed" languages - especially non-expression constructs

Error recovery can be tailored

"Its main principle is that each parser always returns control after having advanced up to a symbol that may legally follow the sentential construct that the parser is supposed to process."

Disadvantages:

Languages with many precedence levels (C++) defy single-symbol lookahead

Pascal precedences simplify grammar, but changes use of parentheses

PL/0:

Allowable beginning symbols for each construct (see railroad diagrams)

```
declbegsys := [constsym, varsym, procsym];              Declarations
statbegsys := [beginsym, callsym, ifsym, whilesym];     Statements
facbegsys  := [ident, number, lparen];                  Factors
```

Follow symbols (right end of construct) accumulate (using Pascal sets) when going deeper into recursion to allow error recovery from:

1. Drastic syntax errors by moving up to a containing construct (`s1` for `test`)
2. Minor errors by skipping over symbols for a contained construct (`s2` for `test`)

Precedence (example of bottom-up)

Infix-to-Postfix (Dijkstra's shunting yard, see http://en.wikipedia.org/wiki/Shunting-yard_algorithm)

Operators - binary, unary, or ?

Parentheses . . . `(` gets pushed, `)` causes pops until matching `(` is found on stack

Identifiers - go immediately to postfix output (e.g. the highest precedence)

Types - checking?

Precedences - stack is ordered ascending from bottom-to-top

Associativity - equal precedence ⇒ operator gets removed from stack

(http://ranger.uta.edu/~weems/NOTES3302/infix2postfix.c)

```
char alpha[]={'(',')','!','*','+','<','&','|','$','#'};

int prec[]=  {20, 30, 90, 80, 70, 60, 50, 40, 0,  10};
```

```
...
void translate()
{
char lastSymbol;  // For detecting improper adjacent symbols

lastSymbol='(';   // Safe way to initialize this
operatorStack[++operatorSP]=0;  // push initial $ to stack
nextSymbol=1;
for (nextSymbol=1; nextSymbol<programSize;nextSymbol++)
{
  checkAdjacentSymbols(lastSymbol,program[nextSymbol]);
  if (program[nextSymbol]>='a' && program[nextSymbol]<='z')
  {
    postfix[postfixLength++]=program[nextSymbol];
    waitingOperands++;
  }
  else if (program[nextSymbol]=='(' || program[nextSymbol]=='!')
    operatorStack[++operatorSP]=nextSymbol;
  else
  {

// Move ripe operators to postfix.  Everything is left-associative
    while (symbol2prec(program[nextSymbol])
          <=symbol2prec(program[operatorStack[operatorSP]]))
    {
      switch(program[operatorStack[operatorSP]])
      {
        case '(':
          printf("Parenthesis mismatch detected at pos %d\n",nextSymbol);
          exit(0);
        case '!':
          if (waitingOperands<1)
          {
            printf("No operands for ! at position %d\n",
              operatorStack[operatorSP]);
            exit(0);
          }
          postfix[postfixLength++]='!';
          break;
        case '*':   case '+':   case '<':   case '&':   case '|':
          if (waitingOperands<2)
          {
            printf("Only %d operands for %c at position %d\n",
              waitingOperands,program[operatorStack[operatorSP]],
              operatorStack[operatorSP]);
            exit(0);
          }
          postfix[postfixLength++]=program[operatorStack[operatorSP]];
          waitingOperands--;
          break;
        default:
          printf("Uncovered case: %c\n",program[operatorStack[operatorSP]]);
          break;
      } // end switch

      operatorSP--;
    } // end while
```

```
    if (program[nextSymbol]==')')
      if (program[operatorStack[operatorSP]]=='(')
        operatorSP--;
      else
      {
        printf(") at position %d doesn't match a (\n",nextSymbol);
        exit(0);
      }
    else
      operatorStack[++operatorSP]=nextSymbol;
  }

  lastSymbol=program[nextSymbol];
} // end for
}


void checkAdjacentSymbols(char first,char second)
{ // Streamlined check on adjacent symbols in input
int firstIsOperand=first==')' || first>='a' && first<='z',
    secondIsOperand=second>='a' && second<='z' || second=='('
                    || second=='!';

if (firstIsOperand==secondIsOperand)
{
  printf("%c followed by %c\n",first,second);
  exit(0);
}
}
```

`$i#`	`i`
`$a+b#`	`ab+`
`$a+b*c+d#`	`abc*+d+`
`$(a+b)*(c+d)<e+f*g|h*i+j<k+l*m*n#`	`ab+cd+*efg*+<hi*j+klm*n*+<|`
`$a|b&c<d+e*!f#`	`abcdef!*+<&|`
`$!a*b+c<d&e|f#`	`a!b*c+d<e&f|`
`$(a+b)(c+d)#`	`) followed by (`
`$(a+b)*(c+d)))#`	`) at position 12 doesn't match a (`
`$(((((a+b)*(c+d)#`	`Parenthesis mismatch detected at pos 16`
`$!!!(a|b&c)&d#`	`abc&|!!!d&`

Also see Lab 2 Fall 2012 for more complete set of operators (including `? :`), type checking, and building expression tree using postfix evaluator/interpreter.