CSE 3302 Notes 2: Syntax

(Last updated 9/12/12 2:23 PM)

2.1. SPECIFYING SYNTAX: REGULAR EXPRESSIONS AND CONTEXT-FREE GRAMMARS

Railroad Diagrams - `http://en.wikipedia.org/wiki/Syntax_diagram`

Also gives format of Backus-Naur Form (BNF) and Extended BNF

Concept of (Context Free) Grammar

Terminal   Nonterminal   Production   Start Symbol

Derivation - leftmost and rightmost

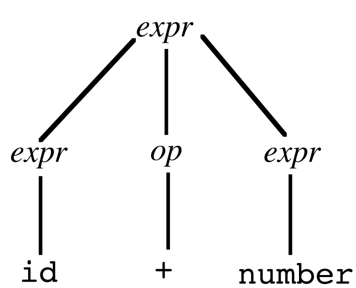
Simple Expression Grammar:

$expr \rightarrow \texttt{id} \mid \texttt{number} \mid \texttt{-}\, expr \mid \texttt{(}\, expr\, \texttt{)} \mid expr\ op\ expr$

$op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$

$expr \Rightarrow expr\ op\ expr \Rightarrow expr + expr \Rightarrow \texttt{id} + expr \Rightarrow \texttt{id} + \texttt{number}$

$expr \overset{+}{\Rightarrow} \texttt{id} + \texttt{number}$



Left and Right Recursion

$x \rightarrow x\ \texttt{a}$

$x \rightarrow \texttt{a}\ x$

ε

Language generated by a grammar

Ambiguity

Classic Expression Grammar

expr → *term* | *expr add_op term*

term → *factor* | *term mult_op factor*

factor → `id` | `number` | `-` *factor* | `(` *expr* `)`

add_op → `+` | `-`

mult_op → `*` | `/`

Regular Expressions

Terminals (*expr*) *expr expr* *expr* | *expr* *expr**

Abbreviations: $expr^k$ $expr^+$

Binary strings: $(0 \mid 1)^+$

Binary strings with even length: $((0 \mid 1)(0 \mid 1))^+$

Deterministic FSAs

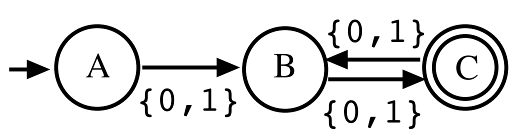
States (start, accepting)

Transitions

(*i*) ε is prohibited on transitions

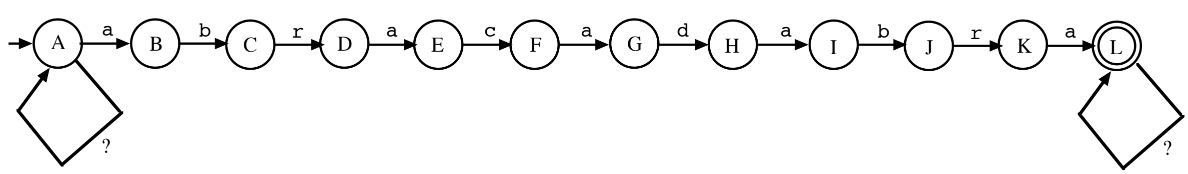
(*ii*) From a given state, two transitions cannot have the same label

Non-empty binary strings with even length:

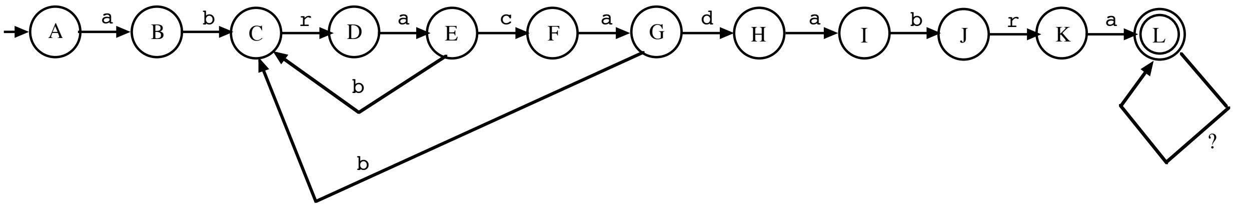


Non-deterministic FSAs (drop (*i*) and (*ii*) above)

Does input string contain `abracadabra`?



Aside: Deterministic counterpart



For each state, except L:

1. If there is no transition shown for `a`, then there is one to B.

2. For all symbols without a transition, there is a transition to A.

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

Languages described by REs, DFSAs, and NDFSAs?

Playing out-of-bounds:

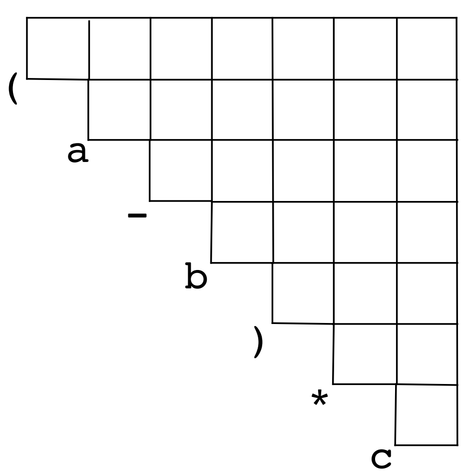
```
collect=/^(.*)(.*)\1{2}\2{3}\1{4}\2{5}$/,result;

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

2.2. SCANNING

2.3. PARSING

Brute Force Recursive (find handles and reduce) - $O\left(n^3\right)$ 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.

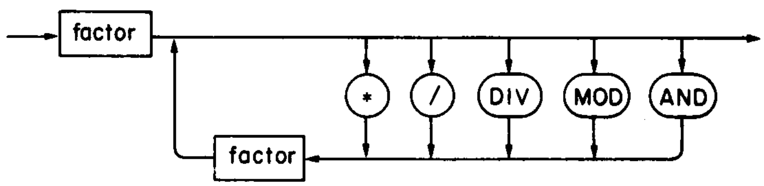
(Asides: Chomsky Normal Form - $a \rightarrow b\ c$ or $a \rightarrow$ b. Definite clause grammars and Prolog)

Similar to DP solution for optimal matrix multiplication

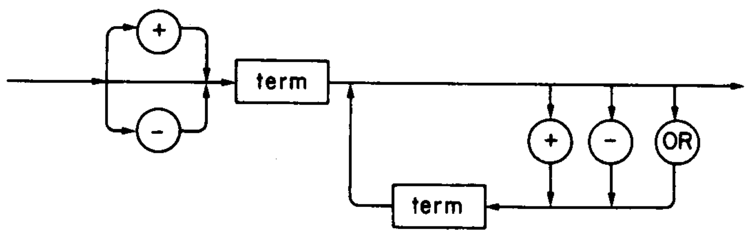
Why is this inefficient?

Recursive Descent (top-down)

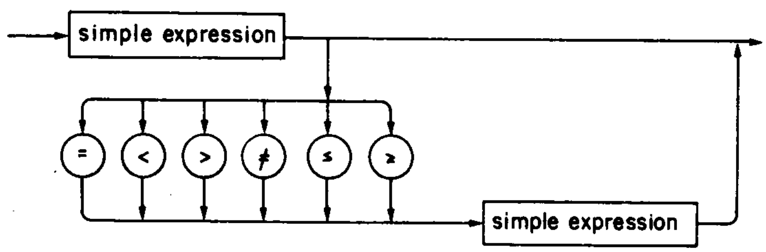
Term



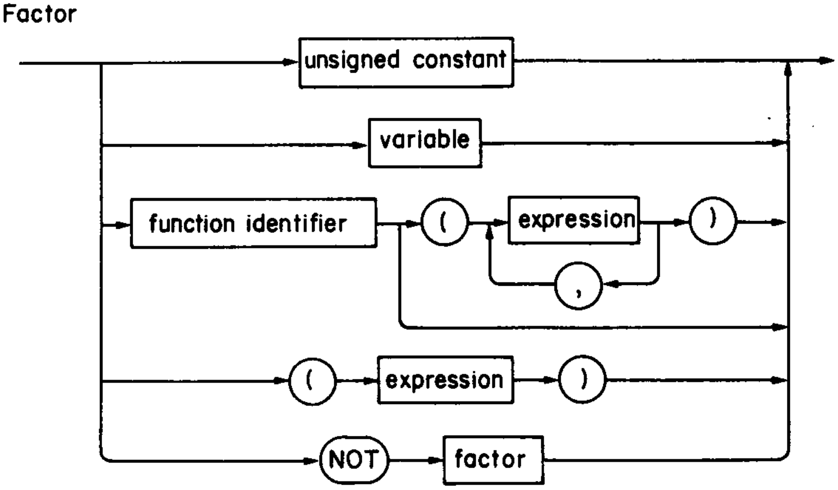
Simple expression



Expression



Factor



```
char* alpha="i+*<&|!()";
int alphaSize;
char program[1000];
int programSize;

typedef enum {
  TERMINAL,SIMPLE,EXPR,TERM,FACTOR,IDENTIFIER,UNKNOWN
} langElement;

typedef struct
{
  langElement element;
  int firstChild,rightSib,parent;  // Multiway tree mapped to binary tree
} nodeType;
nodeType tree[2000];

int nextSymbol; // Simple scan
int nextNode;   // Simple allocation of nodes

// Needed due to recursion (& functions can't nest in C)
int expr(int);
int simple(int);
int term(int);
int factor(int);

int expr(int parent)
// An expression is one or two simple expressions connected by a comparison
{
int myNum=nextNode++,simp1,comparePos,simp2;

tree[myNum].element=EXPR;
tree[myNum].firstChild=simp1=simple(myNum);
// tree[myNum].rightSib is determined elsewhere
tree[myNum].parent=parent;
if (program[nextSymbol]=='<')
{
  comparePos=nextSymbol++;
  tree[simp1].rightSib=comparePos;
  tree[comparePos].element=TERMINAL;
  tree[comparePos].firstChild=(-1);
  tree[comparePos].rightSib=simp2=simple(myNum);
  tree[simp2].rightSib=(-1);
  tree[comparePos].parent=myNum;
}
else
  tree[simp1].rightSib=(-1);
return myNum;
}

int simple(int parent)
// A simple expression is one or more terms connected by additive operators
{
int myNum=nextNode++,term1,plusOrPos,term2;

tree[myNum].element=SIMPLE;
tree[myNum].firstChild=term1=term(myNum);
// tree[myNum].rightSib is determined elsewhere
tree[myNum].parent=parent;
```

```
while (program[nextSymbol]=='+' || program[nextSymbol]=='|')
{
  plusOrPos=nextSymbol++;
  tree[term1].rightSib=plusOrPos;
  tree[plusOrPos].element=TERMINAL;
  tree[plusOrPos].firstChild=(-1);
  tree[plusOrPos].rightSib=term2=term(myNum);
  tree[plusOrPos].parent=myNum;
  term1=term2;
}

tree[term1].rightSib=(-1);
return myNum;
}
```

```
i*i+i<i+i*i
EXPR
  SIMPLE
    TERM
      FACTOR
        IDENTIFIER i
      TERMINAL *
      FACTOR
        IDENTIFIER i
    TERMINAL +
    TERM
      FACTOR
        IDENTIFIER i
  TERMINAL <
  SIMPLE
    TERM
      FACTOR
        IDENTIFIER i
    TERMINAL +
    TERM
      FACTOR
        IDENTIFIER i
      TERMINAL *
      FACTOR
        IDENTIFIER i
```

```
i<i&i*i<i+i
4 unprocessed symbols
EXPR
  SIMPLE
    TERM
      FACTOR
        IDENTIFIER i
  TERMINAL <
  SIMPLE
    TERM
      FACTOR
        IDENTIFIER i
      TERMINAL &
      FACTOR
        IDENTIFIER i
      TERMINAL *
      FACTOR
        IDENTIFIER i
```

```
i<i&(i*i<i+i)
EXPR
  SIMPLE
    TERM
      FACTOR
        IDENTIFIER i
  TERMINAL <
  SIMPLE
    TERM
      FACTOR
        IDENTIFIER i
      TERMINAL &
      FACTOR
        TERMINAL (
        EXPR
          SIMPLE
            TERM
              FACTOR
                IDENTIFIER i
              TERMINAL *
              FACTOR
                IDENTIFIER i
          TERMINAL <
          SIMPLE
            TERM
              FACTOR
                IDENTIFIER i
            TERMINAL +
            TERM
              FACTOR
                IDENTIFIER i
        TERMINAL )
```

Advantages

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

Error recovery can be tailored

Disadvantages:

Languages with many precedence levels (C++)

Pascal precedences to simplify grammar

Precedence (bottom-up)

Attempt to expand Pascal grammar towards C set of operators to observe number/boolean:

aexpr → *aterm* {`+` *aterm*}*

aterm → *afactor* {`*` *afactor*}*

afactor → `id` | `(` *aexpr* `)`

bexpr → *bterm* {`|` *bterm*}*

bterm → *bfactor* {`&` *bfactor*}*

bfactor → `id` | `(` *bexpr* `)` | `!` *bfactor* | *cexpr*

cexpr → *aexpr* {`<` *aexpr*}

Recursive descent needs types of `id`s

Operator-Precedence Parsing (Aho & Ullman, *Principles of Compiler Design*, 1977)

E → *E* + *T* | *T*

T → *T* * *F* | *F*

F → (*E*) | `id`

	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

Bottom-up parse proceeds left-to-right using stack:

<· Beginning of handle ≐ Handle continues ·> End of handle

Develop formally or intuitively?

Do you need grammar parse tree or abstract parse tree or . . . just postfix?

What about associativity?

Infix-to-Postfix (Dijkstra's shunting yard, see wikipedia)

Can construct any of the three representations

Operators - binary, unary, or ?

Parentheses . . .

Identifiers - go immediately to postfix

Types - checking?

Precedences

Associativity

```
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)#                        ) 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&
```