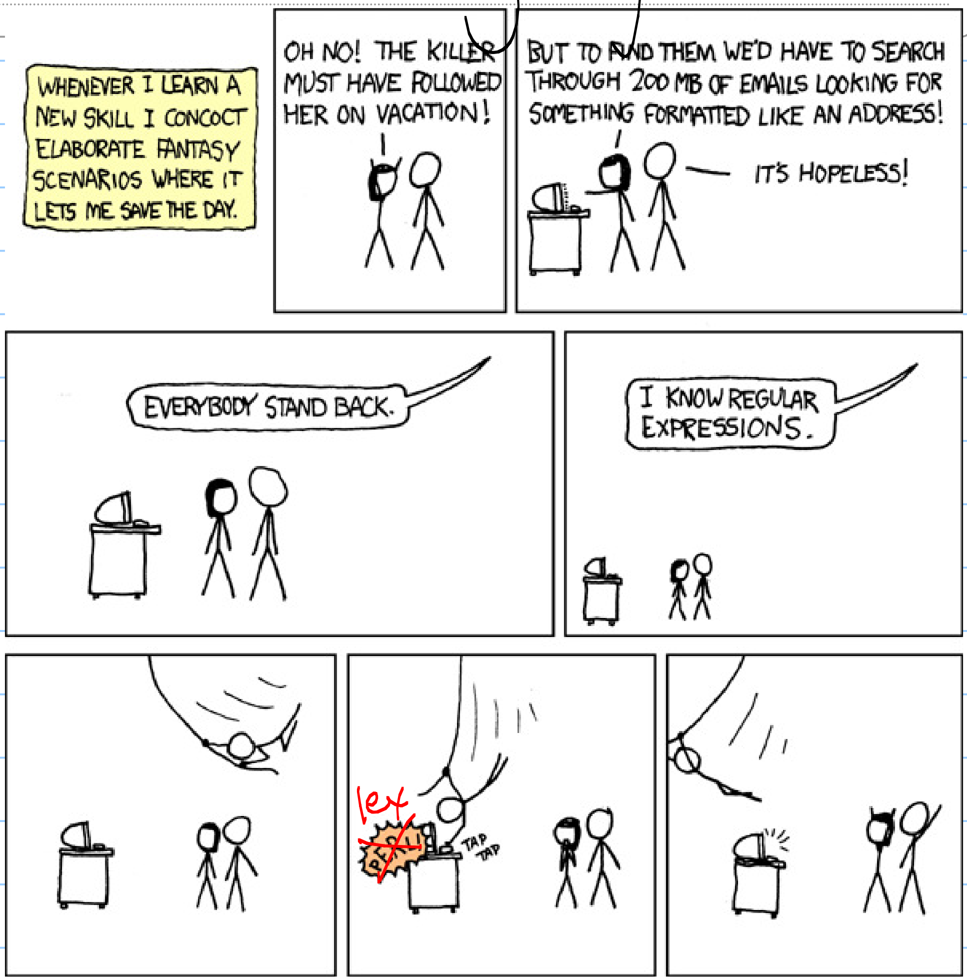
CS 344 - Context Free Languages

Note Title

Announcements

- HW due!
- Next HW up - due next Friday



Last time: flex

A useful scanner.

Based on reg expressions as well as states.

(Examples + links should be posted now.)

Automates building DFA code.

That buggy example:

Well, I was just wrong.

REJECT actually is also for overlapping, so was grabbing subwords:

So:

try␣
ry␣
y␣

} +3 to wc

Another example:

```
%%

pink    { npink++; REJECT; }
ink     { nink++; REJECT; }
pin     { npin++; REJECT; }
. |
\n      ;      /* discard others */
```

So – to fix my word count program:

Avoid REJECT!

Alternate – see .lex file.

Key: add len(word) to charcount
(yylen)

Flex: a tokenizer for a calculator:

```
/* recognize tokens for the calculator and print them out */
%{
   enum yytokentype {
     NUMBER = 258,
     ADD = 259,
     SUB = 260,
     MUL = 261,
     DIV = 262,
     ABS = 263,
     EOL = 264
   };
   int yylval;
%}

%%
"+"    { return ADD; }
"-"    { return SUB; }
"*"    { return MUL; }
"/"    { return DIV; }
"|"    { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n     { return EOL; }
[ \t]  { /* ignore whitespace */ }
.      { printf("Mystery character %c\n", *yytext); }
%%
main(int argc, char **argv)
{
  int tok;

  while(tok = yylex()) {
    printf("%d", tok);
    if(tok == NUMBER) printf(" = %d\n", yylval);
    else printf("\n");
  }
}
```

tokens to hand to parser

In Action:

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
```

Bison accepts these tokens:

```
/* simplest version of calculator */
%{
#include <stdio.h>
%}

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL

%%

calclist: /* nothing */
 | calclist exp EOL { printf("= %d\n", $1); }
 ;

exp: factor         default $$ = $1
 | exp ADD factor { $$ = $1 + $3; }
 | exp SUB factor { $$ = $1 - $3; }
 ;

factor: term        default $$ = $1
 | factor MUL term { $$ = $1 * $3; }
 | factor DIV term { $$ = $1 / $3; }
 ;
```

CFG

```
term: NUMBER   default $$ = $1
 | ABS term   { $$ = $2 >= 0? $2 : - $2; }
;
%%
main(int argc, char **argv)
{
  yyparse();
}

yyerror(char *s)
{
  fprintf(stderr, "error: %s\n", s);
}
```

Building:

```
# part of the makefile
fb1-5:  fb1-5.l fb1-5.y
        bison -d fb1-5.y
        flex fb1-5.l
        cc -o $@ fb1-5.tab.c lex.yy.c -lfl
```

Running:

```
$ ./fb1-5
2 + 3 * 4
= 14
2 * 3 + 4
= 10
20 / 4 - 2
= 3
20 - 4 / 2
= 18
```

Back to what Bison is:

```
exp: factor          default $$ = $1
  | exp ADD factor { $$ = $1 + $3; }
  | exp SUB factor { $$ = $1 - $3; }
  ;

factor: term          default $$ = $1
  | factor MUL term { $$ = $1 * $3; }
  | factor DIV term { $$ = $1 / $3; }
  ;
```

nice

Essentially, this is a CFG!

But only works on a particular type of grammar.

Context-Free Languages

Recall that for any context free languages, there are an infinite # of grammars that can produce it.

We wish to somehow give a definition of a "good" set of productions.

Goal: Parsing (well) –
given a language, detect if a string is in that language.

capitols - non-terminals
lowercase - terminals

Ex: (BAD)

start nonterminal → $S_0 \to S \mid X \mid Z$

$S \to A$

$A \to B$ ← useless - stuck

$C \to Aa$

$X \to C$

unreachable, useless rule → ~~$Y \to aY \mid a$~~

$Z \to \varepsilon$

chain
$S \to X \to C \to Aa$
$\downarrow$
B ...

Goal: avoid ε if possible
avoid $X \to C \to A \to$

Chomsky Normal Form (CNF)

Each rule in the grammar is either:

- $A \to BC$
  where neither B or C is the start variable, & both are nonterminals
- $A \to a$
  where a is a terminal
- $S \to \varepsilon$
  where S is the start symbol

Thm: All grammars can be converted to CNF.

Procedure:

① Create a new start symbol S_0, & send $S_0 \rightarrow S$

(might need $S_0 \rightarrow \varepsilon$)

(1a) Eliminate useless rules

(just delete ones that can't be reached)

② Remove nullable variables.

$A \to \varepsilon$

How?

Remove all ε productions.

Then fix.

Ex: $A \to CBC$

⋮

$B \to \varepsilon \mid b$

$\Rightarrow$

$A \to CBC \mid CC$

$B \to b$

③ Remove unit rules:

$S \to A$

How? Must have:

$X \to Z_1,\ Z_1 \to Z_2,\ \dots,\ Z_k \to Y$

(X, Y) is a unit pair

(since we removed ε-transitions in ②)

Then:

add $X \to Y$

but if $Y \to$ non-term:

For each unit pair (A, B)
and rule $B \to w$,

add $A \to w$ to a <u>new</u>
grammar.

(Note that (A, A) is a unit pair,
so all rules $A \to w$
will stick around.)

$X \to ABCDy$

(4) Get rid of "long" righthand sides.

4a: Create $V_c \to c$ for every character.

Replace c with V_c everywhere.

Now either

$$A \to CDEF$$

or

$$V_c \to c.$$

To demo:

$A \to ABx \mid \varepsilon$

$B \to By \mid \varepsilon$

add start $S_0 \to A \mid \varepsilon$ (step 1)

$A \to \underline{ABx} \mid Bx \mid x \mid Ax$

$B \to By \mid y$

add dummy non-terms:

$S_0 \to A \mid \varepsilon$

$A \to ABx \mid Bx \mid x \mid Ax$

$B \to By \mid y$

New: $S_0 \to A \mid \varepsilon$

$V_x \to x$

$V_y \to y$

$A \to ABV_x \mid BV_x \mid V_x \mid AV_x$

$B \to BV_y \mid V_y$

$A \to CV_x$

$C \to AB$

4b: $A \rightarrow B_1 B_2 B_3 \ldots B_k$

How to replace with only 2 nonterminals on the right?

$A \rightarrow B_1 X_1$

$X_1 \rightarrow B_2 X_2$

$X_2 \rightarrow B_3 X_3$

$\vdots$

~~Unit Pairs: $S_0 \to A$~~   **CNF!**

$S_0 \to \varepsilon \mid CV_x \mid BV_x \mid x \mid AV_x$

$V_x \to x$

$V_y \to y$

$A \to CV_x \mid BV_x \mid x \mid AV_x$

$B \to BV_y \mid y$

$C \to CD \mid BD \mid V_xB \mid AD$

$D \to V_xB$

Ex: Convert:

$S \to ASA \mid aB$

$A \to B \mid S$

$B \to b \mid \varepsilon$

① $S_0 \to S$

$S \to ASA \mid aB \mid a \mid AS \mid SA \mid S$

$A \to \cancel{\varepsilon} \mid S \mid b$

$B \to b$

~~(S_0, S), (A, S)~~

Ex (cont):

~~$S_0 \to S$~~

bSA
ASb
bSb

$S \to AY \mid V_a B \mid a \mid AS \mid SA \mid$ ~~X~~ $\mid XA$

$A \to$ ~~S~~ $\mid b$ $\quad AX \mid XS \mid BY$

$B \to b$ $\quad ZB \mid WB \mid SS$

$\quad BS \mid SB$

$X \to SS$

$Y \to SA \qquad V_a \to a$

$Z \to AS \qquad W \to BS$

Now — why do we care??

Parsing: building those parse trees we saw

In general, there are an exponential number of parse trees for a given input.

So how to check quickly?

Even in CNF, might be 2^n possible parse trees.

Cocke-Younger-Kasami (CYK) algorithm

70's

Uses a table & dynamic programming to give a parse tree in $O(n^3)$ time.

Grammar **must** be in CNF!

5 + 2

num → 5

Other options

- n^3 is still pretty slow.
  - ↑ size of my program

In general, can't really do better.

However, certain classes could be done faster.

- LL(1), LR(1) } $O(n)$ algorithm
  - ↑ lookahead