Announcements

- HW due!

- Next HW up - due next Friday
Last time: flex
A useful scanner.
Based on regex expressions as well as J 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

+ 3 to we

Yes
Another example:

```plaintext
//

pink {
    npink++;
    REJECT;
}

ink {
    nink++;
    REJECT;
}

pin {
    npin++;
    REJECT;
}

in j;  // discard others
```
So - to fix my word count program:
  Avoid REJECT!
  Alternate - see .lex file.

key: add len(word) to charcount
(lvl len)
Flex: a tokenizer for a calculator:

```c
/* recognize tokens for the calculator and print them out */

int yylval;

enum yytokentype {
    NUMBER = 258,
    ADD = 259,
    SUB = 260,
    MUL = 261,
    DIV = 262,
    ABS = 263,
    EOL = 264
}

/* tokens to hand to parser */

[0-9]+ { yylval = atoi(yytext); return NUMBER; }
[ \	] { /* ignore whitespace */ }
[ \
] { return EOL; }

main(int argc, char **argv)
{
    int tok;

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

```
$ flex fb1-4.1
$ 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:

```c
/* 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; }
    ;

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:

```plaintext
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; }
;
```

Essentially, this is a CFG!

But only works on a particular type of grammar.

nice
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.
Ex: (BAD)

```
S₀ → S | X | Z
S → A
A → B
C → Aₐ
X → C
Y → aX|a
Z → ε
```

**Capital - non-terminals**

**Lowercase - terminals**

- **Start non-terminal:** S₀
- **Root non-terminal:** S
- **Useless symbol:** A
- **Useless production:** A → B
- **Unreachable symbol:** C
- **Unreachable production:** C → Aₐ
- **Chain:** S → X → C → Aₐ B

...
**Goal**: avoid ε if possible

*avoid* $X \rightarrow C \rightarrow A$*

**Chomsky Normal Form (CNF)**

Each rule in the grammar is either:

1. $A \rightarrow BC$
   where neither $B$ or $C$ is the start variable, or both are nonterminals

2. $A \rightarrow a$
   where $a$ is a terminal

3. $S \rightarrow ε$
   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 \( S_0 \rightarrow \varepsilon \)).

2. Eliminate useless rules (just delete ones that can't be reached).
2. Remove nullable variables.

\[ A \rightarrow \epsilon \]

How?

Remove all \( \epsilon \) productions.

Then \( A \rightarrow CBC \ | CC \)

\[ B \rightarrow \epsilon \ | b \]

\[ B \rightarrow b \]
3) Remove unit rules:

\[ S \rightarrow A \]

How? Must have:

\[ X \rightarrow z_1, z_1 \rightarrow z_2, \ldots, z_k \rightarrow Y \]

\((X, Y)\) is a unit pair (since we removed \(\varepsilon\)-transitions in \(2\))

Then:

add \(X \rightarrow Y\)

but if \(Y \rightarrow \) non-term:
For each unit pair \((A, B)\) and rule \(B \rightarrow w\),
add \(A \rightarrow w\) to a new grammar.

(Note that \((A, A)\) is a unit pair, so all rules \(A \rightarrow w\) will stick around.)
Get rid of "long" righthand sides.

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

Replace $c$ with $V_c$ everywhere.

Now either

$A \rightarrow CDEF$

or

$V_c \rightarrow c$. 
To demo:

\[ A \rightarrow AB \times \mid \varepsilon \]

\[ B \rightarrow B \gamma \mid \varepsilon \]

add start \[ S_0 \rightarrow A \mid \varepsilon \quad \text{(step 1)} \]

\[ A \rightarrow AB \times \mid B \times \mid \times \mid \varepsilon \]

\[ B \rightarrow B \gamma \mid \gamma \]
add dummy non-terms:

\[ S_0 \rightarrow A \mid \epsilon \]
\[
\begin{align*}
A & \rightarrow AB \mid B \mid x \mid A \mid \\
B & \rightarrow By \mid y
\end{align*}
\]

New:

\[ S_0 \rightarrow A \mid \epsilon \]
\[
\begin{align*}
V_x & \rightarrow x \\
V_y & \rightarrow y
\end{align*}
\]
\[
\begin{align*}
A & \rightarrow ABV_x \mid BV_x \mid V_x \mid AV_x \\
B & \rightarrow BV_y \mid V_y
\end{align*}
\]
46:  \[ A \rightarrow B_1 B_2 B_3 \ldots B_k \]

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 pars: $S_0 \rightarrow A$  \[ \text{CNF!} \]

$S_0 \rightarrow \varepsilon | CV_x | BU_x | x | AV_x$

$V_x \rightarrow x$

$V_y \rightarrow y$

$A \rightarrow CV_x | BU_x | x | AV_x$

$B \rightarrow BV_y | y$

$C \rightarrow CD | BD | V_x | B | A | D$

$D \rightarrow V_x | B$
Ex: Convert:

\[ S \rightarrow \text{ASA} | aB \]

\[ A \rightarrow B | S \]

\[ B \rightarrow b | \varepsilon \]

1. \[ S_0 \rightarrow S \]

\[ S \rightarrow \text{ASA} | aB | a | AS | SA | S \]

\[ A \rightarrow \times S | b \]

\[ B \rightarrow b \]
\[(S_0, S), (A, S)\]

Ex. (cont.):

\[\text{Ex} \]

\[S \rightarrow AY - | \quad \text{wb} \quad | a \quad \text{AS} | \text{SA} | X | X A\]

\[A \rightarrow X | b \quad \text{AX} | \text{XS} | \text{BY}\]

\[B \rightarrow b \quad \text{ZB} | \text{WB} | \text{SS}\]

\[X \rightarrow SS \quad \text{BS} | \text{SB}\]

\[Y \rightarrow SA \quad V_a \rightarrow a\]

\[Z \rightarrow AS \quad W \rightarrow BS\]
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

Uses a table & dynamic programming

to give a parse tree in $O(n^3)$ time.

Grammar must be in CNF!
Other options
- $n^3$ is still pretty slow.

Site of my program

In general, can't really do better.

However, certain classes could be done faster.

- $LL(1), LR(1)$ is $O(n)$ algorithm

Look ahead