CS3200 — Scanning & flex

Note Title 1/27/2012

Announcements

- Essay due Wed.
- No class Mon or Wed
- HW2 is posted, due Sat Jan 30

Compilers

Front end:

(A) Scanner

(B) Parser

(C) Semantic Analysis

We're focusing on scanners first...

Scanning (lexical analysis)

- Divide program into tokens, or smallest meaningful units

  Ex: for, recognize keywords, group operations, names, etc.

- Scanning & tokenizing makes parsing much simpler.

- While parsers can work character by character, it is slow.

- Note: Scanning is recognizing a regular language, eg via DFA

Last time: Scanners recognize:

- regular expressions (or reg. languages)
- can also form as DFA or NFA (more next week on these)
- Some limitations: can't do anything with memory:

  Ex: $0^n 1^n$ exp w/ equal # of (&)

But: good for tokenizing! (later we'll parse)

Scanners: do this in code

Find the syntax (not semantics) of code.

Output tokens.

A few types:

- Ad-hoc
- Finite automata
  - nested case statements
  - table + driver

Ad-hoc: case based code

```
if current ∈ { "(", ")", "+", "-", "*" }
      return that symbol
if current = ":"
      read next
        if it is =, announce "assign"
              else announce error
if   current = "/"
        read next
        if it is "*" or "/"
              read until "*/" or "newline" (resp.)
        else return divide
etc.
```

Ad-hoc approach

Advantage:
code is fast & compact

Disadvantage:
very ad-hoc!
- hard to debug
- no explicit representation

primary disadvantage

DFA approach:

Given a regular expression, convert to a DFA.

We'll walk through this next week - in Ch 2 of the book.

However →

Scanning Programs

In reality, this DFA is often done automatically.

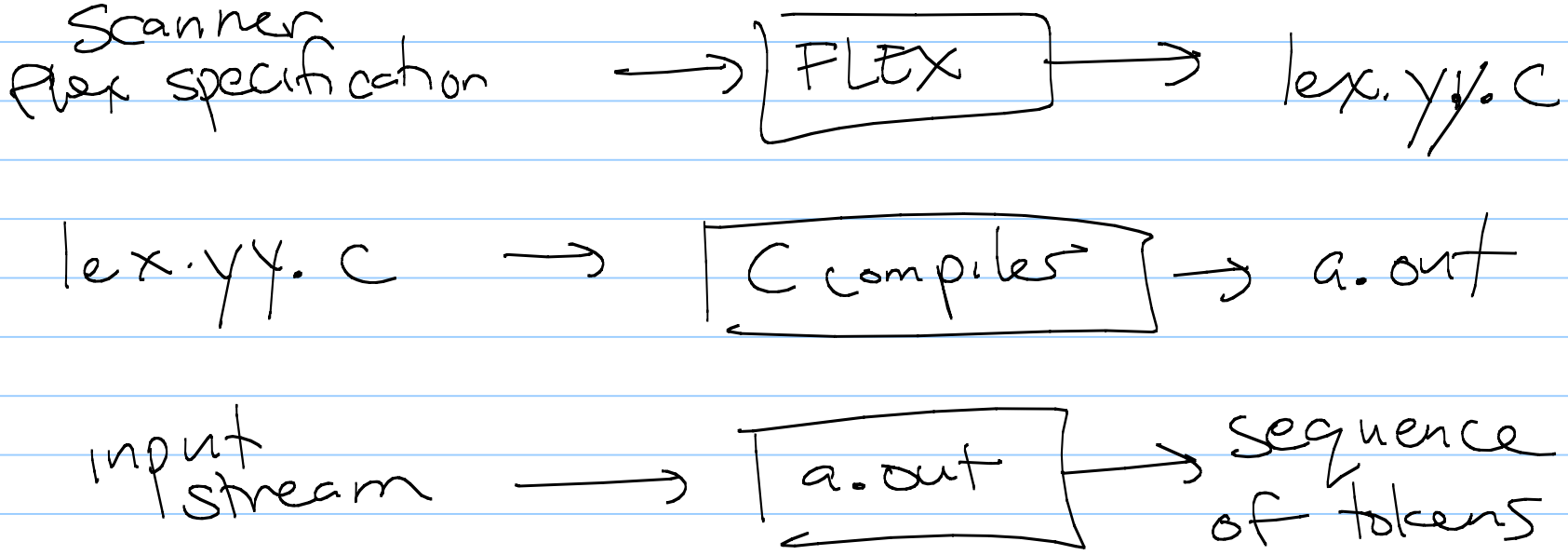
Specify the rules of regular language, & the program does this for you.

Many such examples:

Lex (flex), JLex / JFlex, Quex, Ragel, ...

Flex

- A C driven scanning program.



To compile:

```
> flex file.lex

> gcc lex.yy.c -lfl
> ./a.out (< otherfile.txt)
                 if input
```

(if using stdin, might need ctl-d)

Format for .lex files:

1. definitions

%%

2. rules

%%

user code

(see examples)

1 Definitions

New definitions to make life easier.

Form: name definition

Ex:

digit [0-9]

ID [a-z] [a-zA-Z0-9]*

[A-Z] also

Note: These are regular expressions!

Definitions cont:

- An unidented comment (/*) is copied verbatim to output, up to the next */
- Any indented text or text enclosed in: %{ ... }% is also copied verbatim (with %{ ... }% removed)
- %top makes sure things are copied to top of output (for example, for #includes)

2 Rules Section

Format: pattern action

where pattern is unindented, action is on the same line

Any intendented or %{ %} can be used to declare variables. local to the scanning routine.

(Other things may cause compile issues)

Allowed Patterns

'x' – match the character x

'.' – any char except newline

'[x y z]' – matches x, y, or z

'[ab j-o Z]' – matches a, b, j, k, l, m, n, o, Z

More patterns

`[^A-Z]` - chars other than A-Z (negation)

^ = not

`[^A-Z\n]` - any char except A-Z or a newline

`[a-z]{-}[aeiou]` - any lower case Consenant

0 or more

1 or more

r^*, r^+

Patterns (again)

`'r?'` 0 or 1 r's

`'r{2-5}'` Between 2 & 5 r's

`'r{2,}'` 2 or more r's

`'r{4}'` exactly 4 r's

`'{name}'` expansion of name definition

`'r$'` r at end of a line

:

(post webpage)

Precendence:

foo | bar*
is same as (foo) | (ba(r)*)

(since * has higher precedence than concatenation, + Concatenation is higher than or)

C classes

[:alpha:] matches anything that satisfies .isalpha(y).

Ex:

[[:alnum:]]

[[:alpha:][:digit:]]

[[:alpha:][0-9]]

[[a-zA-Z][0-9]]

3 User code

Optional, just copied directly to the output.

(if empty, leave off 1st %%)

Comments

- C style : /* */

Exceptions Do: pattern rule /* comment */

- No comments in the rule section when a regular expression is expected (so not beginning of line or after scanner states)

- Not on %option line of definitions

How it works

- Finds longest pattern match possible
- That match (or token) is made available to a global char pointer yytext w/ length in yyleng
  Then action is performed
- If no match, next char goes to std out.
  (So
  %%
  is valid.)

Actions

Ex:

```
%%
"zap me"
```

Ex:

```
%%
[ \t]+
[ \t]+$    putchar(' ');
           /* ignore */
```

Actions (cont)

- If action contains a {, then action spans until next } (and may go over many lines)

- Action | means "same action as the next rule"

- Can be arbitrary C code, including a return. (When run again continues from where it left off.)

Special Actions

- ECHO

- BEGIN followed by name of a start condition places scanner in that condition (more on this later...)

- REJECT tells scanner to go to second best rule

  ↳ caution: slow

Ex: Count the # of words

pattern to look for:

character [a-z A-Z']

character* [' ./n!]

Ex:

```
%%

a       |
ab      |
-abc    |
abcd    ECHO; REJECT;
```

Scans: xyzabcd

Output? xyz abcd abc ab a abcd

Conditional Rules

- State based! activated using BEGIN

Define a set of states
- INITIAL is there by default
- Rest defined in %s or %x in first section

Ex:

```
%s STRING

%%
<STRING>[^"]*    BEGIN(STRING);
<STRING>"        {action;}
                 BEGIN(INITIAL)
```

%s are inclusive start conditions
%x are exclusive start conditions

After BEGIN, state is active.

If state is inclusive, then rules with no start conditions are still active.

If state is exclusive, then rules with no start conditions are inactive.

Ex: %s versus %x

```
%s example
%%
<example>foo    action();
bar             other_action();
```

Vs:

```
%x example
%%
<example>foo    action();
<INITIAL,example>bar    other_action();
```

Conditions

- <*> matches all states
- default rule is in all states.

  Essentially, pretend:

  <*>./\n ECHO;

  is a line of your file.

Ex: Scanner to ignore C comments & count of current input line

```
%x comment

%%

        int num_line = 1;

"/*"    BEGIN(comment);

<comment>[^*\n]*
<comment>"*"+[^*/\n]

<comment>\n     ++line_num;
<comment>"*"+"/"    BEGIN(INITIAL);
```

Can condense

<comment> {

all rules

}