CS 536 Announcements for Wednesday, February 7, 2024

Programming Assignment 2

- released later today
- due Tuesday, February 20

Last Time

- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

Today

- JLex
- CFGs

Next Time

- CFG ambiguity

JLex

Declarative specification : you don't tell JLex how to scan / how to match tokens, you tell JLex what you want scanned (tokens) & what to do when a token is matched

Input: set of regular expressions + associated actions
Output: Java source code for a scanner → jlex.java contains Yylex

Format of JLex specification : 3 sections separated by %%

- user code section
- directives
- regular expression rules

Example

```
// User Code section:  For right now, we will not use it.
%%
```

Directives

```
DIGIT=        [0-9]
LETTER=       [a-zA-Z]
WHITESPACE=   [\040\t\n]
```

macro definitions
format: name = regular expression
space, tab, newline
\040 = ASCII/Unicode for space

```
%state SPECIALINTSTATE
```

← state declaration

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
```

needed to use generated scanner with Java Cup

```
%eofval{
System.out.println("All done");
return null;
%eofval}
```

tell JLex what to do on EOF

```
%line
```

← turn on line counting (starts at 0)

```
%%
({LETTER}|"_")({DIGIT}|{LETTER}|"_")* {
                          System.out.println(yyline+1 + ": ID "
                                    + yytext()); }

"="              { System.out.println(yyline+1 + ": ASSIGN"); }
"+"              { System.out.println(yyline+1 + ": PLUS"); }
"^"              { System.out.println(yyline+1 + ": EXP"); }
"<"              { System.out.println(yyline+1 + ": LESSTHAN"); }
"+="             { System.out.println(yyline+1 + ": INCR"); }
"<="             { System.out.println(yyline+1 + ": LEQ"); }
{WHITESPACE}*    { }
.                { System.out.println(yyline+1 + ": bad char"); }
```

Regular expression rules section

Format: `<regex>{code}` where `<regex>` is a regular expression for a single token

- can use macros from Directives section – surround with curly braces { }
- characters represent themselves (except special characters)
- characters inside " " represent themselves (except \")
- . matches anything

Regular expression operators: `|  *  +  ?  ( )`

Character class operators: `-  ^  \`

Using scanner generated by JLex in a program

```
// inFile is a FileReader initialized to read from the
// file to be scanned

Yylex scanner = new Yylex(inFile);

try {

    scanner.next_token();

} catch (IOException ex) {
    System.err.println(
                "unexpected IOException thrown by the scanner");
    System.exit(-1);
}
```

Why regular expressions are not good enough

Regular expression wrap-up

\+ perfect for tokenizing a language

– limitations

- define only limited family of languages
  - can't be used to specify all the programming constructs we need
- no notion of structure

Regexs cannot handle "matching"

Example: $L_{()} = \{ (^n)^n \text{ where } n > 0\}$ = { "( )", "(( ))", "((( )))" .. }

Theorem: No regex/DFA can describe the language $L_{()}$

Proof by contradiction: Suppose there exists a DFA *A* for $L_{()}$ where *A* has *N* states.

Then *A* has to accept the string $(^N)^N$ with some sequence of states

$$q_0 \xrightarrow{(} q_1 \xrightarrow{(} q_2 \xrightarrow{(} \cdots \xrightarrow{(} q_N \xrightarrow{)} q_{N+1} \xrightarrow{)} q_{N+2} \xrightarrow{)} \cdots \xrightarrow{)} q_{2N+1}$$

accepts $(^N$ accepts $)^N$

By the pigeonhole principle, there exists $i, j \le N$ where $i < j$ such that $q_i = q_j$

So

$$q_0 \xrightarrow{(} q_1 \xrightarrow{(} q_2 \xrightarrow{(} \cdots q_{i=j} \xrightarrow{(} q_{j+1} \xrightarrow{(} \cdots \xrightarrow{(} q_N \xrightarrow{)} q_{N+1} \xrightarrow{)} q_{N+2} \xrightarrow{)} \cdots \xrightarrow{)} q_{2N+1}$$

accepts $($ $i+(N-j)$ accepts $)^N$

In other words, A accepts $(^{N-j+i})^N$ but $(^{N-j+i})^N \notin L_{()}$

which is a contradiction

No notion of structure

Consider the following stream of tokens: ID ASSIGN ID PLUS ID (x = y + z)

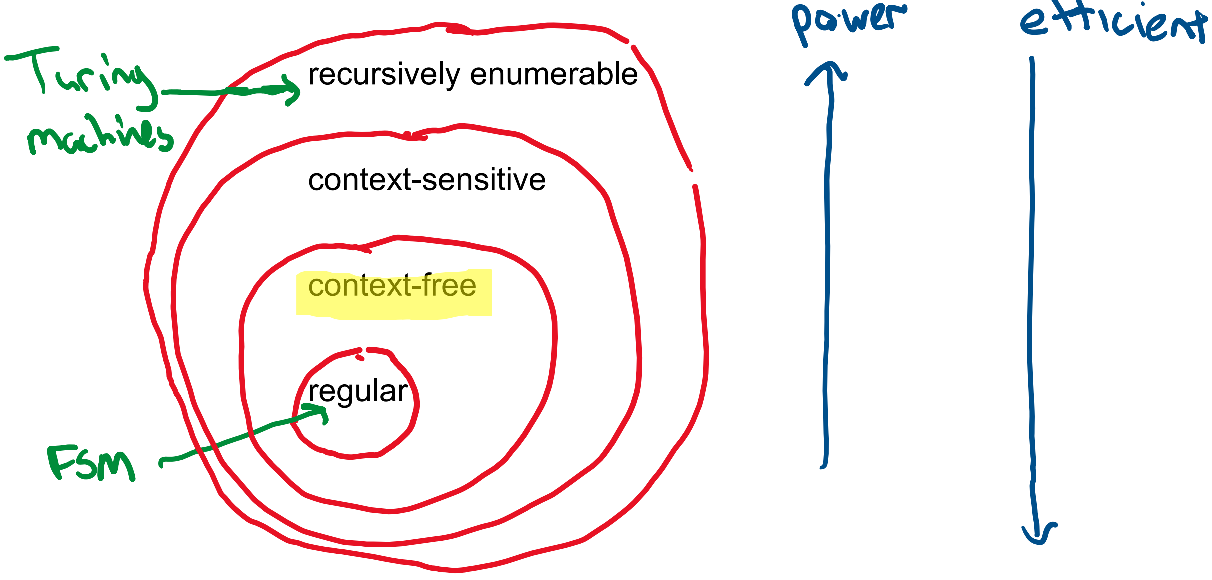
What should be done 1st? $(x=y)+z$ or $x=(y+z)$

What about precedence & associativity?

(Noam)

The Chomsky Language Hierarchy

Language class:



Context-free grammar (CFG)

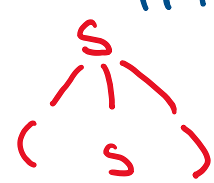
= a set of recursive rewriting rules to generate patterns of strings

S → '(' S ')'

rule for rewriting S

Before applying: S

After applying:



Formal definition: A CFG is a 4-tuple (N, ∑, P, S)

- N = set of **non-terminals** – placeholders (interior nodes in parse tree)
- ∑ = set of **terminals** – tokens from scanner
- P = set of **productions** – rules for re-writing non-terminals (for deriving)
- S = initial non-terminal symbol ("start symbol"), S ϵ N

– if not otherwise specified, use non-term on LHS of 1st production as start symbol

Productions

Production syntax : LHS → RHS

LHS ↑ single non-terminal

RHS ↑ expression (seq of terms & non-terms) or ε

non-term → expression $S \to$ '(' S ')'

non-term → ε $S \to \varepsilon$

or (assuming non-terms on LHS are the same)

non-term → expression $S \to$ '(' S ')'

| ε | ε

or

non-term → expression | ε $S \to$ '(' S ')' | ε

Language defined by a CFG

= set of strings (i.e., sequences of terminals) that can be derived from the start non-terminal

To derive a string (of terminal symbols):

- set Curr_Seq to start symbol
- repeat
  - find a non-terminal x in Curr_Seq
  - find production of the form x → α
  - "apply" production: create new Curr_Seq by replacing x with α
- until Curr_Seq contains no non-terminals

Derivation notation

- derives $\Rightarrow$
- derives in one or more steps $\Rightarrow^+$ or $\stackrel{+}{\Rightarrow}$
- derives in zero or more steps $\Rightarrow^*$ or $\stackrel{*}{\Rightarrow}$

L(G) = language defined by CFG G

= $\{ w \mid S \stackrel{+}{\Rightarrow} w$ where S is start non-term & w is a seq of terms or ε $\}$

↑ "such that"

Example grammar

Terminals

BEGIN
END } – program boundary

SEMICOLON – ";" to separate statements

ASSIGN – "=" in assignment stmts

ID – identifier (variable name)

PLUS – "+" operator in expression

Non-terminals

prog – start non-term (root of parse-tree)

stmts – list of statements

stmt – a single statement

expr – a (mathematical) expression

Productions – define syntax of legal programs

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID

↑ #s for reference only

Example derivation

Productions

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID

Derivation

prog ⟹ BEGIN stmts END   using ①

⟹ BEGIN stmts SEMICOLON stmt END   using ②

⟹ BEGIN stmt SEMICOLON stmt END   using ③

⟹ BEGIN ID ASSIGN expr SEMICOLON stmt END   using ④

⟹ BEGIN ID ASSIGN expr SEMICOLON ID ASSIGN expr END   using ④

⟹ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr END   using ⑤

⟹ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr PLUS ID END   using ⑥

⟹ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN ID PLUS ID END   using ⑤

or BEGIN ID = ID ; ID = ID + ID END

prog $\overset{+}{\Longrightarrow}$ BEGIN ID = ID ; ID = ID + ID END

Parse trees

= way to visualize a derivation

To derive a string (of terminal symbols):

- set root of parse tree to start symbol
- repeat
  - find a leaf non-terminal x
  - find production of the form x → α
  - "apply" production: symbols in α become the children of x
- until there are no more leaf non-terminals

Derived sequence determined from leaves, from left to right

Productions

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID