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

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.

%%

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

%state SPECIALINTSTATE

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

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

%line
```

```
%%

({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: $L_{(\,)} = \{\, (^n)^n$ 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

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

So

In other words,

No notion of structure

Consider the following stream of tokens: ID ASSIGN ID PLUS ID

The Chomsky Language Hierarchy

Language class:

recursively enumerable

context-sensitive

context-free

regular

Context-free grammar (CFG)

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

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

- N = set of **non-terminals**
- ∑ = set of **terminals**
- P = set of **productions**
- S = initial non-terminal symbol ("start symbol"), S ϵ N

Productions

Production syntax : LHS → RHS

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
- derives in one or more steps
- derives in zero or more steps

L(G) = language defined by CFG G

=

Example grammar

Terminals

BEGIN

END

SEMICOLON

ASSIGN

ID

PLUS

Non-terminals

prog

stmts

stmt

expr

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

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

⟹ BEGIN stmts SEMICOLON stmt END

⟹ BEGIN stmt SEMICOLON stmt END

⟹ BEGIN ID ASSIGN expr SEMICOLON stmt END

⟹ BEGIN ID ASSIGN expr SEMICOLON ID ASSIGN expr END

⟹ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr END

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

⟹ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN ID PLUS 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