03-60-214 Jlex

Inside lexical analyzer generator

- How does a lexical analyzer work?
  - Get input from user who defines tokens in the form that is equivalent to regular grammar
  - Turn the regular grammar into a NFA
  - Convert the NFA into DFA
  - Generate the code that simulates the DFA

JLex

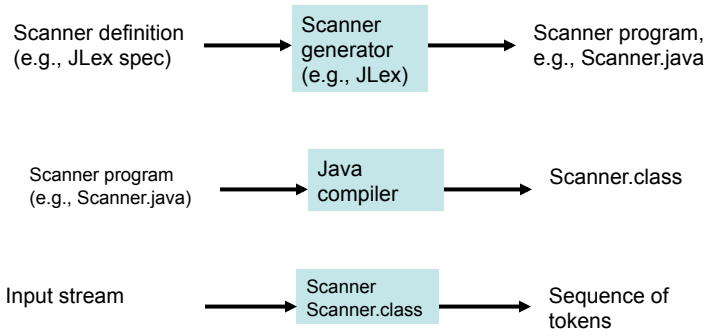
Classes in JLex:

- CAccept
- CAcceptAnchor
- CAlloc
- CBunch
- CDfa
- CDTrans
- CEmit
- CError
- CInput
- CLexGen
- CMakeNfa
- CMinimize
- CNfa
- CNfa2Dfa
- CNfaPair
- CSet
- CSimplifyNfa
- CSpec
- CUtility
- Main
- SparseBitSet
- ucsb

How scanner generator is used

JLex

- Write the scanner specification;
- Generate the scanner program using scanner generator;
- Compile the scanner program;
- Run the scanner program on input streams, and produce sequences of tokens.



3

JLex specification

JLex

- JLex specification consists of three parts, separated by "%%"

User Java code, to be copied verbatim into the scanner program, placed before the lexer class;

%%

JLex directives,
macro definitions, commonly used to specify letters, digits, whitespace;

%%

Regular expressions and actions:

- Specify how to divide input into tokens;
- Regular expressions are followed by actions;
  - Print error messages; return token codes;

4

JLex

First JLex example simple.lex

- Recognize int and identifiers.

```
1.  %%
2.  %{   public static void main(String argv[]) throws java.io.IOException {
3.       MyLexer yy = new MyLexer(System.in);
4.       while (true){
5.         yy.yylex();
6.        }
7.       }
8.  %}
9.  %notunix
10. %type void
11. %class MyLexer
12. %eofval{ return;
13. %eofval}

14. IDENTIFIER = [a-zA-Z][a-zA-Z0-9]*

15. %%

16. "int" { System.out.println("INT recognized");}
17. {IDENTIFIER} { System.out.println("ID is ..." + yytext());}
18. \r|\n {}
19. . {}
```

5

JLex

Code generated will be in simple.lex.java

```
class MyLexer {
    public static void main(String argv[]) throws java.io.IOException {
        MyLexer yy = new MyLexer(System.in);
        while (true){
          yy.yylex();
        }
    }
    public void yylex(){
     ... ...
      case 5:{ System.out.println("INT recognized"); }
      case 7:{ System.out.println("ID is ..." + yytext()); }
      ... ...
    }
}
```

Copied from internal code directive

6

Running the JLex example

JLex

- Steps to run the JLex

```
D:\214>java JLex.Main simple.lex
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 22 states.
Working on character classes.:::::::::.
NFA has 10 distinct character classes.
Creating DFA transition table.
Working on DFA states.....
Minimizing DFA transition table.
9 states after removal of redundant states.
Outputting lexical analyzer code.

D:\214>move simple.lex.java MyLexer.java

D:\214>javac MyLexer.java

D:\214>java MyLexer      // it is waiting for keyboard input
int myid0
INT recognized
ID is ...myid0
```


Exercises

JLex

- Try to modify JLex directives in the previous JLex spec, and observe whether it is still working. If it is not working, try to understand the reason.
  - Remove "%notunix" directive;
  - Change "return;" to "return null;";
  - Remove "%type void";
  - ... ...
- Move the Identifier regular expression before the "int" RE. What will happen to the input "int"?
- What if you remove the last line (line 19, ". {}") ?

Change simple.lex: read input from file

JLex

```
1.  import java.io.*;
2.  %%
3.  %{   public static void main(String argv[]) throws java.io.IOException {
4.     MyLexer yy = new MyLexer( new FileReader("input") );
5.     while (yy.yylex()>=0);
6.     }
7.  %}
8.  %integer
9.  %class MyLexer

10. %%

11. "int" { System.out.println("INT recognized");}
12. [a-zA-Z_][a-zA-Z0-9_]* { System.out.println("ID is ..." + yytext());}
13. \r|\n|. {}
```

- %integer: to make the returning type of yylex() as int.

9

Extend the example: add returning and use classes

JLex

– When a token is recognized, in most of the case we want to return a token object, so that other programs can use it.

```
class UseLexer {
  public static void main(String [] args) throws java.io.IOException {
    Token t;   MyLexer2 lexer=new MyLexer2(System.in);
     while ((t=lexer.yylex())!=null) System.out.println(t.toString());
  }
}
class Token {
  String type;   String text;     int line;
  Token(String t, String txt, int l) { type=t; text=txt; line=l; }
  public String toString(){ return text+" " +type + " " +line; }
}
%%
%notunix
%line
%type Token
%class MyLexer2
%eofval{ return null;
%eofval}
IDENTIFIER = [a-zA-Z_][a-zA-Z0-9_]*
%%
"int" { return(new Token("INT", yytext(), yyline));}
{IDENTIFIER} { return(new Token("ID", yytext(), yyline));}
\r|\n {}
. {}
```

10

Code generated from mylexer2.lex

JLex

```
class UseLexer {
  public static void main(String [] args) throws java.io.IOException {
    Token t;   MyLexer2 lexer=new MyLexer2(System.in);
     while ((t=lexer.yylex())!=null) System.out.println(t.toString());
  }
}
class Token {
  String type;   String text;     int line;
  Token(String t, String txt, int l) { type=t; text=txt; line=l; }
  public String toString(){ return text+" " +type + " " +line; }
}

Class MyLexer2 {
  public Token  yylex(){
     ... ...
      case 5: { return(new Token("INT", yytext(), yyline)); }
      case 7: { return(new Token("ID", yytext(), yyline)); }
                ... ...
   }

}
```


Running the extended lex specification mylexer2.lex

JLex

```
D:\214>java JLex.Main mylexer2.lex
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 22 states.
Working on character classes.:::::::::.
NFA has 10 distinct character classes.
Creating DFA transition table.
Working on DFA states.....
Minimizing DFA transition table.
9 states after removal of redundant states.
Outputting lexical analyzer code.

D:\214>move mylexer2.lex.java MyLexer2.java

D:\214>javac MyLexer2.java

D:\214>java UseLexer
int
int INT 0
x1
x1 ID 1
```

Another example

JLex

```
1    import java.io.IOException;
2    %%
3    %public
4    %class  Numbers_1
5    %type   void
6    %eofval{ return;
8    %eofval}
9
10   %line
11   %{     public static void main (String args []) {
12     Numbers_1 num = new Numbers_1(System.in);
13     try {
14       num.yylex();
15     } catch (IOException e) { System.err.println(e); }
16    }
17   %}
18
19   %%
20   \r\n     { System.out.println("--- " + (yyline+1));  }
22   .*\r\n    { System.out.print  ("+++ " + (yyline+1)+"\t"+yytext());  }
```

13

User code (first section of JLex)

JLex

- User code is copied verbatim into the lexical analyzer source file that JLex outputs, at the top of the file.
  - Package declarations;
  - Imports of an external class
  - Class definitions
- Generated code

```
package declarations;
import packages;
Class definitions;
class Yylex {
   ... ...
}
```

- Yylex class is the default lexer class name. It can be changed to other class name using *%class* directive.

14

JLex directives (Second section)

JLex

- Internal code to lexical analyzer class
- Marco definition
- State declaration
- Character/line counting
- Lexical analyzer component title
- Specifying the return value on end-of-file
- Specifying an interface to implement

Internal Code to Lexical Analyzer Class

JLex

- %{ …. %} directive permits the declaration of variables and functions internal to the generated lexical analyzer
- General form:

```
%{
    <code >
%}
```

- Effect: <code > will be copied into the Lexer class, such as MyLexer.

```
class MyLexer{
  ..... <code> ......
}
```

- Example

```
public static void main(String argv[]) throws java.io.IOException {
    MyLexer yy = new MyLexer(System.in);
      while (true){   yy.yylex();      }
}
```

- Difference with the user code section
  - It is copied inside the lexer class (e.g., the MyLexer class)

Macro Definition

JLex

- Purpose: define once and used several times;
  - A must when we write large lex specification.
- General form of macro definition:
  - <name> = <definition>
  - should be contained on a single line
  - Macro name should be valid identifiers
  - Macro definition should be valid regular expressions
  - Macro definition can contain other macro expansions, in the standard {<name>} format for macros within regular expressions.
- Example
  - Definition (in the second part of JLex spec):

    IDENTIFIER = [a-zA-Z_][a-zA-Z0-9_]*

    ALPHA=[A-Za-z_]

    DIGIT=[0-9]

    ALPHA_NUMERIC={ALPHA}|{DIGIT}
  - Use (in the third part):

    {IDENTIFIER} {return new Token(ID, yytext()); }

17

State directive

JLex

- Same string could be matched by different regular expressions, according to its surrounding environment.
  - String "int" inside comment should not be recognized as a reserved word, not even as an identifier.
- Particularly useful when you need to analyze mixed languages;
- For example, in JSP, Java programs can be imbedded inside HTML blocks. Once you are inside Java block, you follow the Java syntax. But when you are out of the Java block, you need to follow the HTML syntax.
  - In java "int" should be recognized as a reserved word;
  - In HTML "int" should be recognized just as a usual string.
- States inside JLex

  <HTMLState> %{ { yybegin(JavaState); }

  <HTMLState> "int" {return string; }

  <JavaState> %} { yybegin(HTMLState); }

  <JavaState> "int" {return keyword; }

18

State Directive (cont.)

JLex

- Mechanism to mix FA states and REs
- Declaring a set of "start states" (in the second part of JLex spec)
  %state state0 [, state1, state2, ….]
- How to use the state (in the third part of JLex spec):
  - RE can be prefixed by the set of start states in which it is valid;
- We can make a transition from one state to another with input RE
  - yybegin(STATE) is the command to make transition to STATE;
- YYINITIAL : implicit start state of yylex();
  - But we can change the start state;
- Example (from the sample in JLex spec):

```
%state COMMENT
%%
<YYINITIAL>if          {return new tok(sym.IF,"IF");}
<YYINITIAL>[a-z]+      {return new tok(sym.ID, yytext());}
<YYINITIAL>"/*"        {yybegin(COMMENT);}
<COMMENT>"*/"          {yybegin(YYINITIAL);}
<COMMENT>.             {}
```

19

Character and line counting

JLex

- Sometimes it is useful to know where exactly the token is in the text. Token position is implemented using line counting and char counting.
- Character counting is turned off by default, activated with the directive "%char"
  - Create an instance variable `yychar` in the scanner;
  - zero-based character index of the first character on the matched region of text.
- Line counting is turned off by default, activated with the directive "%line"
  - Create an instance variable yyline in the scanner;
  - zero-based line index at the beginning of the matched region of text.♪
- Example

  "int" { return (new Yytoken(4,yytext(),yyline,yychar,yychar+3)); }

20

Lexical analyzer component titles

JLex

- Change the name of generated
  - lexical analyzer class %class <name>
  - the tokenizing function %function <name>
  - the token return type %type <name>
- Default names

```
class Yylex {      /* lexical analyzer class */
  public Yytoken          /* the token return type */
          yylex() { …}    /* the tokenizing function */
  ==> Yylex.yylex()  returns  Yytoken  type
```

21

Specifying an Interface to implement

JLex

- Form: %implements <InterfaceName>
- Allows the user to specify an interface which the Yylex or your lexer class will implement.
- The generated parser class declaration will look like:

```
class MyLexer implements InterfaceName {
 …….
}
```

22

Regular expression rules

JLex

- General form: regularExpression { action}
- Example: {IDENTIFIER} { System.out.println("ID is ..." + yytext());}
- Interpretation: Patten to be matched code to be executed when the pattern is matched
- Code generated in MyLexer:

" case 2: { System.out.println("ID is ..." + yytext());} "

23

Regular Expression Rules

JLex

- Specifies rules for breaking the input stream into tokens
- Regular Expression + Actions (java code)
  [<states>] <expression> { <action>}
- When matched with more than one rule,
  - choose the rule that is given first in the Jlex spec.
    - Refer the "int" and IDENTIFIER example.
- The rules given in a JLex specification should match all possible input.
- An error will be raised if the generated lexer receives input that does not match any of its rules
  - E.g., the rules only listed the case for Identifiers, and said nothing about numbers, but your input has numbers.
  - This is the most common error (more than 50%)
  - put the following rule at the bottom of RE spec
    - `{java.lang.System.out.println("Error:" + yytext());}`

  dot(.) will match any input except for the newline.

24

Available lexical values within action code

JLex

- java.lang.String yytext()
  - matches portion of the character input stream;
  - always active.
- Int yychar
  - Zero-based character index of the first character in the matched portion of the input stream;
  - activated by *%char* directive.
- Int yyline
  - Zero-based line number of the start of the matched portion of the input stream;
  - activated by *%line* directive.

25

Regular expression in JLex

JLex

- Special characters: ? + | ( ) ^ $ / ; . = < > [ ] { } " \ and blank
  - After \ the special characters lose their special meaning.
  - Example: \+
- Between double quotes " all special characters but \ and " lose their special meaning.
  - Example: "+"
- The following escape sequences are recognized: \b \n \t \f \r.
- With [ ] we can describe sets of characters.
  - [abc] is the same as (a|b|c). Note that it is not equivalent to abc
  - With [^] we can describe sets of characters.
  - [^\n\"] means anything but a newline or quotes
  - [^a–z] means anything but ONE lower-case letter
- We can use . as a shortcut for [^\n]
- $: denotes the end of a line. If $ ends a regular expression, the expression matched only at the end of a line.

26

Concluding remarks

- Focused on Lexical Analysis Process, Including
  - Regular Expressions
  - Finite Automaton
  - Conversion
  - Lex
- Regular grammar=regular expression
- Regular expression→ NFA→ DFA→ lexer
- The next step in the compilation process is Parsing:
  - Context free grammar;
  - Top-down parsing and bottom up parsing.

JLex

27