CS 461

Programming Language Concepts

Gang Tan
Computer Science and Engineering
Penn State University

The parsing is divided into two steps

- First step: lexical analysis (lexer, scanner)
  - Convert a sequence of chars to a sequence of tokens
  - Token: a logically cohesive sequence of characters
  - Common tokens
    - Identifiers
    - Literals: 123, 5.67, "hello", true
    - Keywords: bool char ...
    - Operators: + - * / ++ ...
    - Punctuation: ; , ( ) { }
- Second step: syntactic analysis (parser)
  - Convert a sequence of tokens into an AST

Regular Expressions

- Used extensively in languages and tools for pattern matching
  - E.g., Perl, Ruby, grep
- Regular expression operations
  - ϵ (pronounced as epsilon) matches the empty string: epsilon
  - a, a literal character, matches a single character
  - Alternation: r1 | r2
    - e.g., 0|1|…|9,
  - Concatenation: r1 r2
    - e.g: (a|b) c
  - Repetition (zero or more times, Kleene star): r*
    - **e.g: a***

Extended Regular Expressions

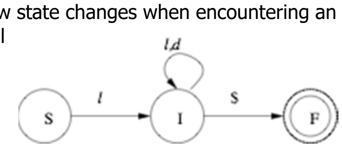
- One or more repetitions
  - r+: digit+ where digit = 0|1|…|9
- Zero or one occurrence: r?
  - E.g., a?
- A set of characters: [aeiou]
- A range of characters in the alphabet
  - a|b|c: [abc]
  - a|b|…|z:[a-z]
  - 0|1|…|9: [0-9]
- Q: How to encode the above constructs using operators in regular expressions?

Lexical Analysis

- Purpose: transform program representation
- Input: a sequence of printable characters
- Output: a sequence of tokens
- Also
  - Discard whitespace and comments
  - Save source locations (file, line, column) for error messages

Finite State Automata

- A finite set of states
  - Unique start state
  - One or more final states
    - Drawn in double circles
- Input alphabet + unique end symbol ($)
- State transition function: T[s,c]
  - Describe how state changes when encountering an input symbol



FSA Execution

- An input is *accepted* if, starting with the start state, the automaton consumes all the input and halts in a final state.

```
s = startState;
while( s not in finalState) {
  c = next_input_character;
  s = T[s,c];
}
```

Examples: xx0$, x12$; non-examples: 0x$

The language recognized by an FSA is the set of input strings accepted by the FSA

7

Deterministic FSA

- Defn: A finite state automaton is *deterministic* if for each state, there are no two outgoing edges labelled with the same input character
- A deterministic FSA gives a way of recognizing a language
- Theorem: for each RE, we can construct a deterministic FSA that recognizes the language of the RE

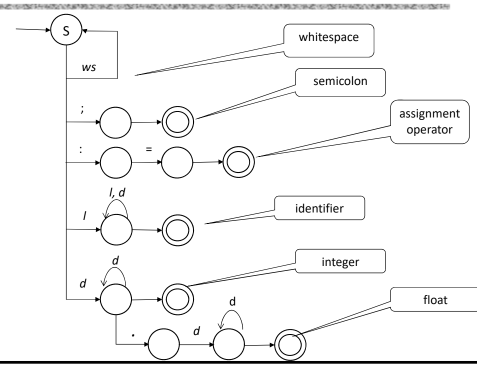
8

A Running Example for Lexer and Parser

- A statement language in E-BNF
  - <stmt> -> <assignment> {;<assignment>}
  - <assignment> -> <id> := <exp>
  - <exp> -> <id> | <int> | <float>
  - Tokens:
    - <id>=<letter>(<letter>|<digit>)*
    - <int> = <digit>+
    - <float> = <digit>+.<digit>+
    - punctuation marks: ; , :=, $

9

DFA for the Running Example



Constructing a Lexer: Token.java

```
public class Token {
    public enum TokenType {INT, FLOAT, ID, SEMICOLON,
    ASSIGNMENTOP, EOI, INVALID}

    private TokenType type;
    private String val;

    Token (TokenType t, String v) { type = t; val = v; }
    TokenType getTokenType() {return type;}
    String getTokenValue() {return val;}

    void print () { … }

     …

}
```

11

The Structure of Lexer.java

```
class Lexer{

   String stmt;
   int index = 0; // index is the index to the next input character
   char ch; // ch is the current character

    // initialization code
   public Lexer(String s){stmt = s; index=0; ch = nextChar();}

    // nextToken() returns the next available token
   public Token nextToken() {
      do {
         …
      } while (true);
   }
   …
}
```

12

Lexer: nextToken(), part I

```
public Token nextToken() {
    do {
        if (Character.isLetter(ch)) {
            String id = concat (letters + digits);
            return new Token(Token.TokenType.ID, id);
        } else if (Character.isDigit(ch)) {
            String num = concat(digits);
            if (ch != '.')
                return new Token(Token.TokenType.INT, num);
            num += ch; ch = nextChar();
            if (Character.isDigit(ch)) {
                num += concat(digits);
                return new Token(Token.TokenType.FLOAT, num);
            } else return new Token(Token.TokenType.INVALID, num);
        } else switch (ch) {
            …
        }
    } while (true);
}
```

13

Lexer: nextToken(), part II

```
public Token nextToken(){
    do {
        if  (…) { …
        } else switch (ch) {
            case ' ':
                ch = nextChar(); break;
            case ';':
                ch = nextChar();
                return new Token(Token.TokenType.SEMICOLON, "");
            case ':':
                if (check('='))
                    return new Token(Token.TokenType.ASSIGNMENTOP, "");
                else return new Token(Token.TokenType.INVALID, ":");
            case '$':
                return new Token(Token.TokenType.EOI, "");
            default:
                ch = nextChar();
                return new Token(Token.TokenType.INVALID,
                                 Character.toString(ch));
        }
    } while (true);
}
```

14

Some Aux. Functions for the Lexer

```
private char nextChar() {
    char ch = stmt.charAt(index); index = index+1;
    return ch;
}

private boolean check (char c) {
    ch = nextChar();
    if (ch == c) {ch = nextChar(); return true;}
    else return false;
}

private String concat (String set) {
    StringBuffer r = new StringBuffer("");
    do { r.append(ch); ch = nextChar();
    } while (set.indexOf(ch) >= 0);
    return r.toString();
}
```

15

An Example of Running the Lexer

```
lexer = new Lexer ("x := 1; y := x $");
tk = lexer.nextToken();
while (tk.getTokenType() != Token.TokenType.EOI) {
    tk.print(); System.out.print(" ");
    tk = lex.nextToken();
}
```

16

Recursive descent parsing

- Implementation follows directly the BNF grammar
  - <stmt> -> <assignment> {;<assignment>}
  - <assignment> -> <id> := <exp>
  - <exp> -> <id> | <int> | <float>
- Each non-terminal comes with a parser method
  - statement(); assignmentStmt(); expression();
  - Usually a parser method returns an object of corresponding class
    - E.g., expression() should return an expression object and statement() should return a statement object
  - The code we show next, however, just prints out the parse tree

17

Parser Method for Statements

```
public void statement () {
    System.out.println("<Statement>");
    assignmentStmt();
    while (token.getTokenType() == Token.TokenType.SEMICOLON) {
        System.out.println("\t<Semicolon>;</Semicolon>");
        token = lexer.nextToken();
        assignmentStmt();
    }
    match(Token.TokenType.EOI);
    System.out.println("</Statement>");
}
```

<stmt> -> <assignment> {;<assignment>}

18

Parser Method for Assignment

```
public void assignmentStmt () {
    System.out.println("\t<Assignment>");
    String val = match(Token.TokenType.ID);
    System.out.println("\t\t<Identifier>" + val + "</Identifier>");
    match(Token.TokenType.ASSIGNMENTOP);
    System.out.println("\t\t<AssignmentOp>:=</AssignmentOp>");
    expression();
    System.out.println("\t</Assignment>");
}
```

<assignment> -> <id> := <exp>

19

Parser Method for Expression

```
public void expression () {
    if (token.getTokenType() == Token.TokenType.ID) {
        System.out.println("\t\t<Identifier>" + token.getTokenValue()
                                + "</Identifier>");
    } else if (token.getTokenType() == Token.TokenType.INT) {
        System.out.println("\t\t<Int>" + token.getTokenValue()
                                + "</Int>");
    } else if (token.getTokenType() == Token.TokenType.FLOAT) {
        System.out.println("\t\t<Float>" + token.getTokenValue()
                                + "</Float>");
    } else {
        System.err.println("Syntax error: expecting a ID, an int, or a float");
        System.exit(1);
    }
    token = lexer.nextToken();
}
```

<exp> -> <id> | <int> | <float>

20

Auxiliary Method for the Parser

```
private String match (Token.TokenType tp) {
        String value = token.getTokenValue();
        if (token.getTokenType() == tp)
            token = lexer.nextToken();
        else error(tp);
        return value;
}
```

21