CS 461

Programming Language Concepts

Gang Tan
Computer Science and Engineering
Penn State University

1

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

2

2

Regular Expressions

- Used extensively in languages and tools for pattern matching
  - E.g., Perl, Ruby, grep
- Regular expression operations
  - ² (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***

3

3

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?

4

4

Lexical Analysis

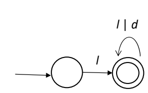
- Purpose: transform program representation
- Input: a sequence of printable characters
- Output: a sequence of tokens
- Also
  - Discard whitespace and comments
  - Save source locations (file, line, column) for error messages

5

5

Finite State Automata

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



6

6

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 (next_char_exists()==true) {
  c = next_char(); s = T[s,c];
}
accept the input iff s in finalStates
```

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

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

7

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

8

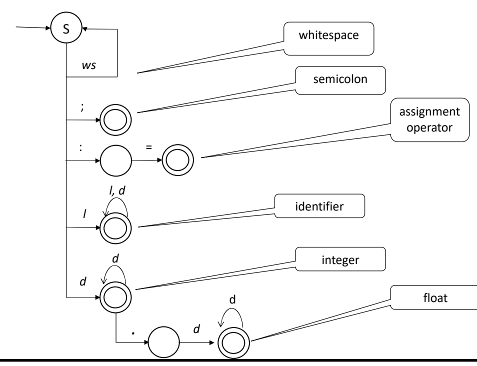
A Running Example for Lexer and Parser

- 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: ; , :=, $
  - Assume the input program always ends with a special end-of-input symbol: $

9

9

DFA for the Running Example



10

Constructing a Lexer: Token class

```
INT, FLOAT, ID, SEMICOLON, ASSIGNMENTOP, EOI, INVALID = 1, 2, 3, 4, 5, 6, 7

LETTERS = "abcdefghijklmnopqrstuvmxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
DIGITS = "0123456789"

class Token:
    # a Token object has two fields: the token's type and its value
    def __init__ (self, tokenType, tokenVal):
        self.type = tokenType
        self.val = tokenVal

    def getTokenType(self):   return self.type
    def getTokenValue(self):   return self.val

    # define the behavior when printing a Token object
    def __repr__(self):  …
```

11

11

The Structure of Lexer

```
class Lexer:

    # stmt is the current statement to perform the lexing;
    # index is the index of the next char in the statement
    def __init__ (self, s):
        self.stmt = s
        self.index = 0
        self.nextChar()

    def nextChar(self):
        self.ch = self.stmt[self.index]
        self.index = self.index + 1

    # nextToken() returns the next available token
    def nextToken(self):
            while True:
                    …
    …
```

12

12

Lexer: nextToken(), part I

```
def nextToken(self):
    while True:
        if self.ch.isalpha(): # is a letter
            id = self.consumeChars(LETTERS+DIGITS)
            return Token(ID, id)
        elif self.ch.isdigit():
            num = self.consumeChars(DIGITS)
            if self.ch != ".":
                return Token(INT, num)
            num += self.ch
            self.nextChar()
            if self.ch.isdigit():
                num += self.consumeChars(DIGITS)
                return Token(FLOAT, num)
            else: return Token(INVALID, num)
        elif …
```

13

13

Lexer: nextToken(), part II

```
def nextToken(self):
    while True:
        if …
        elif self.ch==' ': self.nextChar()
        elif self.ch==';':
            self.nextChar()
            return Token(SEMICOLON, "")
        elif self.ch==':':
            if self.checkChar("="):
                return Token(ASSIGNMENTOP, "")
            else: return Token(INVALID, "")
        elif self.ch=='$':
            return Token(EOI,"")
        else:
            self.nextChar()
            return Token(INVALID, self.ch)
```

14

14

Some Aux. Functions for the Lexer

```
def nextChar(self):
    self.ch = self.stmt[self.index]
    self.index = self.index + 1

def consumeChars (self, charSet):
    r = self.ch
    self.nextChar()
    while (self.ch in charSet):
        r = r + self.ch
        self.nextChar()
    return r

def checkChar(self, c):
    self.nextChar()
    if (self.ch==c):
        self.nextChar()
        return True
    else: return False
```

15

15

An Example of Running the Lexer

```
lex = Lexer ("x := 1; y:=x $")
tk = lex.nextToken()
while (tk.getTokenType() != EOI):
    print tk
    tk = lex.nextToken()
print
```

16

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

17

Parser Method for Statements

```
def statement(self):
    print "<Statement>"
    self.assignmentStmt()
    while self.token.getTokenType() == SEMICOLON:
        print "\t<Semicolon>;</Semicolon>"
        self.token = self.lexer.nextToken()
        self.assignmentStmt()
    self.match(EOI)
    print "</Statement>"
```

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

18

18

Parser Method for Assignment

```
def assignmentStmt(self):
    print "\t<Assignment>"
    val = self.match(ID)
    print "\t\t<Identifier>" + val + "</Identifier>"
    self.match(ASSIGNMENTOP)
    print "\t\t<AssignmentOp>:=</AssignmentOp>"
    self.expression()
    print "\t</Assignment>"
```

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

19

19

Parser Method for Expression

```
def expression(self):
    if self.token.getTokenType() == ID:
        print "\t\t<Identifier>" + self.token.getTokenValue() \
            + "</Identifier>"
    elif self.token.getTokenType() == INT:
        print "\t\t<Int>" + self.token.getTokenValue() + "</Int>"
    elif self.token.getTokenType() == FLOAT:
        print "\t\t<Float>" + self.token.getTokenValue() + "</Float>"
    else:
        print "Syntax error: expecting an ID, an int, or a float" \
            + "; saw:" + typeToString(self.token.getTokenType())
        sys.exit(1)
    self.token = self.lexer.nextToken()
```

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

20

20

Auxiliary Method for the Parser

```
def match (self, tp):
    val = self.token.getTokenValue()
    if (self.token.getTokenType() == tp):
        self.token = self.lexer.nextToken()
    else: self.error(tp)
    return val
```

21

21