CS 461

Programming Language Concepts

Gary 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

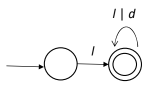
- Purpose: transform program representation
- Input: a sequence of printable characters
- Output: a sequence of tokens
- Also
  - Discard whitespace and comments
  - Save source locations (file, line, column) for error messages

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

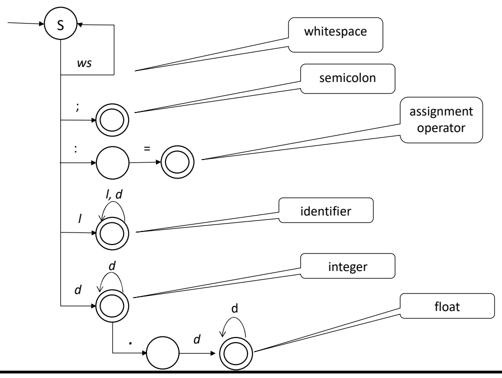
A Running Example for Lexer and Parser

- 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 =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
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==':':
            self.nextChar()
           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 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):
   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

The Parser Code

```
class Parser:
   def __init__(self, s):
      self.lexer = Lexer(s+"$")
      self.token = self.lexer.nextToken()

   def run(self):
      self.statement()

   def statement(self):  …

   def assignmentStmt(self): …

   def expression(self):  …
```

18

18

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>}

19

19

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>

20

20

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>

21

21

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
```

22

22

Lexer and Parser Generators

- Lexer generators: From regular expressions to lexer code
  - C/C++: Lex, Flex
    - Lex by Mike Lesk and Eric Schmidt
  - Java: JLex
- Parser generators: From CFG to parser code
  - C: yacc
  - Table-driven instead of using recursive descent parsing
  - The downside: sometimes generate unreadable code
- We will discuss one particular generator based on a variation of recursive descent parsing
  - Before that, we discuss the limitation of recursive descent parsing

23

Left Recursion Trouble in Recursive Descent

<exp> → <exp> + <term> | <exp> - <term> | <term>

- If naively following recursive descent parsing,

void exp () { exp(); … }

Resulting in infinite loop!

24

24

Left Recursion Removal

- Rewrite the grammar in a different form

```
      <exp> -> <term> {(+ | -) <term>}

void exp() {
   term();
   while (token==plus_op || token == minus_op) {
        token = nextToken();
        term();
   }
}
```

25

25

Left Recursion Removal In General

$\langle A\rangle \rightarrow \langle A\rangle\alpha_1 \mid \ldots \mid \langle A\rangle\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

converted to

$\langle A\rangle \rightarrow \beta_1\langle R\rangle \mid \ldots \mid \beta_m\langle R\rangle$

$\langle R\rangle \rightarrow \alpha_1\langle R\rangle \mid \ldots \mid \alpha_n\langle R\rangle \mid \varepsilon$

26

26

PEG (Parsing Expression Grammars)

- PEG parsing solve this limitation
  - PEGs are similar to CFGs
    - Difference: ordered choice
    - A -> r1 | r2;   if r1 succeeds, PEG parsing won't try the second
  - Packrat: efficient implementation
    - A variation of recursive descent
    - Using memoisation: linear-time parsing; accommodates even left recursion
- Demo using the TatSu library
  - https://tatsu.readthedocs.io/en/stable/
  - Input: a PEG grammar
  - Output: a parser

27

27

TatSu Demo for the Example Language (first version)

```
# specifies a name of the grammar; the parser generator uses this as
# the base name of the generated parser classes
@@grammar::STMT

# "$" specifies the end of the input
start = assignment {';' assignment}* $ ;
assignment = identifier ':=' exp ;
exp = identifier | float | integer ;

# regular expressions are put into "/ ... /"
identifier = /([a-z]|[A-Z])([a-z]|[A-Z]|[0-9])*/ ;
integer = /[0-9]+/ ;
float = /[0-9]+\.[0-9]+/ ;
```

{…}* means 0-or-more repetition

Terminal strings in quotes

28

28

Ordering in Choices Important

exp = identifier | float | integer ;

- Important to put float before integer
  - Because of the ordered choice semantics of PEGs
  - If "exp = identifier | integer | float",
    - Then for "5.23", it would parse 5 as an integer and then leave ".23" as the remaining input
  - In general, need to put longest choice earlier in choices

29

29

Constructing the Parser and Testing

```
def simple_parse():
   grammar = open('stmt.ebnf').read()
   parser = tatsu.compile(grammar)
   try:
      ast = parser.parse('x := 1; y:=x')
      print('# JSON')
      print(json.dumps(ast, indent=4))
   # catch all exceptions
   except tatsu.exceptions.ParseError as e:
      print("syntax error", e)
```

30

30

Semantic Values

- The previous generated parser returns a parse tree that keeps all information in the input
- But it’s unnecessary
  - E.g., once the parser constructs a list of assignments, there is no need to remember the separator is “;”
- A typical parser
  - Throws away information that is unnecessary for the following phases
  - In parsing terminology, they apply “semantic actions” on the parse tree to get “semantic values” that capture essential parsing results

31

31

Semantic Values

- For the example parser, the ideal resulting data structure after parsing
  - Just need to remember a list of assignments, and for each remembers the left-hand side (lhs) and right-hand side (rhs)
- However, the previous parser return oddly shaped results
  - “x:=1” treated differently from “y:=x”

32

32

Annotated Version: Produce Better Semantic Values

```
@@grammar::STMT

# ";" specifies the separator between assignments
# { ... }+ means 1 or more repetition
start = ';'.{assignment}+ $ ;

# lhs (left hand side) gives a label to the identifier to make it
# easier to read
assignment = lhs:identifier ':=' rhs:exp ;

exp = identifier | float | integer ;

identifier = /([a-z]|[A-Z])([a-z]|[A-Z]|[0-9])*/ ;
integer = /[0-9]+/ ;
float = /[0-9]+\.[0-9]+/ ;
```

Separation info is thrown away

No need to remember “:=“

33

33