Midterm Review

October 10, 2019

Covered topics

- Overview
- Grammar
- Arithlang
- Varlang
- Funclang

Important knowledge and problem solving skills

Types of questions

1. Questions about understanding concepts:
   - overview and grammar
   - Arithlang, Varlang, Funclang: understand the grammar and semantic rules
   - ArithLang concepts: prefix, infix and postfix
   - VarLang concepts: define and use a variable, free/bound variable, hole, environment
2. Grammar problem solving:
   - string, grammar, parsing
   - grammar understanding and analysis, ambiguity
   - grammar construction
3. Programming:
   - Arithlang, Varlang: implementing different semantics in interpreter
   - ArithLang, VarLang and FuncLang programming: high order functions, recursive functions, curried form

Overview

What is a Programming Language

A language that can express all computation:

- syntax: validity - is the string a valid program of a programming language (does the string conform to the the grammar of the programming language)?
- semantics: meaning - how to generate a value from the string?

Parts of the Programming Languages

- Computation: to actually compute, e.g., atomic operators
- Composition: to put together computation, e.g., loops, branches . . .
- Abstraction: to achieve scalable programming, e.g., functions, variables

Classification of Programming Languages

- General Purpose languages: C, Java, Scheme ...
- Domain Specific languages: html, dot, sql ...

- High level language: human programs - python, Java, C programs
- Assembly language
- Machine language: computer executes - binary program

Programming Paradigms

Ways of thinking about computation:

- Imperative programming: steps
- object-oriented: objects (some textbooks include this to imperative programming paradigms)
- Functional programming: functions
- Logic programming: facts and relations

Sample Questions:

- (4 pt) Which of the following is/are true about program paradigms?
  (a) functional programming paradigm treats computation as mathematical functions and pure functional programming languages are side-effect free
  (b) the logic programming languages are suitable for programming AI systems
  (c) recursion is a feature that only functional programming paradigm supports
  (d) domain specific languages are imperative programming languages

Sol a b

- (4 pt) Which of the following is/are true about programming languages?
  (a) a programming language is a language that expresses the computation
  (b) a programming language consists of atomic computation, composition and abstraction
  (c) you only need grammar and semantic rules to implement a programming language
  (d) practical programming languages use mostly context free grammar rules to specify the syntax

Sol a b c d

Grammar

Specify patterns in string

a program is a string a program is a string that follows certain "rules";

the rules can be specified by:

- informal grammar – English description
- formal grammar
  1. what are the atoms? (**terminals**, **lexeme**, **token**), e.g., int a = 0; lexeme: int, a, =, 0, ; terminals (given in the grammar) identifier, numbers; tokens (terminal + link to the symbol table, for implementing compiler or interpreter)
  2. how to compose them to form a sentence?
  3. others, e.g., regular expressions

a program is **syntactically correct** (**valid**) if it follows the grammar of the language (rules) in which it is written

Formal Grammars Define Formal Languages

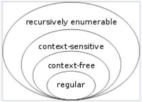
Chomsky Hierarchy

The following table summarizes each of Chomsky's four types of grammars, the class of language it generates, the type of automaton that recognizes it, and the form its rules must have.

Grammar	Languages	Automaton	Production rules (constraints)*	Examples[3]
Type-0	Recursively enumerable	Turing machine	$\alpha A \beta \rightarrow \gamma$	$L = \{w \| w$ describes a terminating Turing machine$\}$
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$	$L = \{a^n b^n c^n \| n > 0\}$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \alpha$	$L = \{a^n b^n \| n > 0\}$
Type-3	Regular	Finite state automaton	$A \rightarrow \mathrm{a}$ and $A \rightarrow \mathrm{a}B$	$L = \{a^n \| n \geq 0\}$

* Meaning of symbols:

- a = terminal
- A, B = non-terminal
- α, β, γ = string of terminals and/or non-terminals
  - α, β = maybe empty
  - γ = never empty



Set inclusions described by the Chomsky hierarchy

Context Free Grammar (CFG)

- 4-tuple: start symbol, terminals, non-terminals, production rules
- Start symbol: represent the program
- terminals: lexeme
- non-terminals: units that compose the program (invisible in the code, abstract), e.g., Digit, Exp; consider that a sentence has sub-components
- production rules: the rules to derive from a non-terminal (what a unit consists of)

Derivation

Derive a string from the start symbol by applying a set of production rules

- there are different ways to derive a string: left-most derivation, right-most derivation and others.
- what if you cannot find a derivation for the string? Syntax errors in the string
- if the grammar is non-ambiguous, you generate only one parse tree independent of how you derive the string

Parsing and Parse Tree

Generate a parse tree from a string

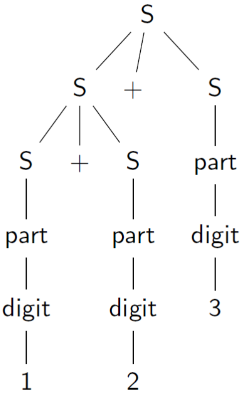
- Start symbol is the root
- Parent is the non-terminal, its children are the terminals or non-terminals on the right hand side of a production rule you select at the current step of the derivation
- Terminals are leaves and non-terminals are internal nodes
- If we cannot find a parse tree for a given string, the string does not belong to the language

Parsing and Parse Tree: Example

Given a grammar,
digit$\rightarrow$ 0|1|2|3|4|5|...|9
part$\rightarrow$ digit|digit part
S$\rightarrow$ part|S+S |S-S|S*S|S/S

find the derivation for 1+2+3

Parsing and Parse Tree: Example (contd.)



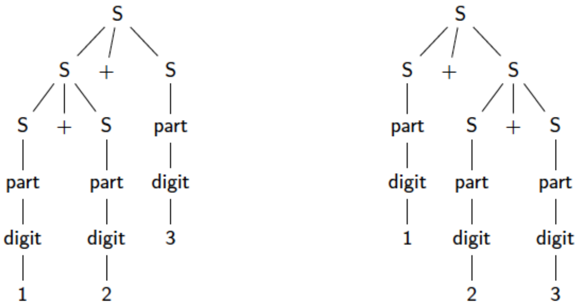
The same parse tree can be generated by left-most and right-most derivation.

Ambiguity

- bad
- can generate different parse trees for the same string - thus has different meanings for the same string
- relations of derivations and parse trees:
  - there are two choices to determine a derivation: 1) which non-terminal to replace, 2) which production rule of the same non-terminal to choose
  - left/right most derivation may generate the same or different parse trees (dependent on which production rules to choose at each step)
  - the left and right most derivations may generate the same parse tree even the grammar is ambiguous and even for the strings that can have two parse trees

Ambiguity (contd.)

Parse Tree for 1 + 2 + 3



Approaches for eliminating ambiguity

Modify grmmar rules:

- Delimiters (e.g., parenthesis) - need to add terminals
- Precedence for operators (intuitively, the operators that have higher priorities should be located in the lower part of the parse tree, and, thus further from the start symbol in the grammar rule)
- Associativity (the current operand should compute with the left or right operand, grammar: allow expansion for the left side, thus, we should repeat the left-hand-side non-terminal on the left)

  $$\mathrm{S} \rightarrow S + A$$

You can also keep the ambiguous grammar rules, but add additional rules via English descriptions

Designing Grammars

1. Use recursive productions to generate an arbitrary number of symbols

   $A \rightarrow xA \mid \varepsilon$ // Zero or more x's

   $A \rightarrow yA \mid y$ // One or more y's

2. Use separate non-terminals to generate disjoint parts of a language, and then combine in a production

   a*b* // a's followed by b's

   $S \rightarrow AB$

   $A \rightarrow aA \mid \varepsilon$ // Zero or more a's

   $B \rightarrow bB \mid \varepsilon$ // Zero or more b's

Note. The superscripted */+ used in the regular expressions are not multiplication or addition, e.g., a^* means zero or more a's, a^+ means one or more a's.

Designing Grammars

3. To generate languages with matching, balanced, or related numbers of symbols, write productions which generate strings from the middle

   $\{a^n b^n \mid n \geq 0\}$ // N a's followed by N b's

   $S \rightarrow aSb \mid \varepsilon$

   Example derivation: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$

   $\{a^n b^{2n} \mid n \geq 0\}$ // N a's followed by 2N b's

   $S \rightarrow aSbb \mid \varepsilon$

   Example derivation: $S \Rightarrow aSbb \Rightarrow aaSbbbb \Rightarrow aabbbb$

Sample Questions:

3. (10 pt) Grammar understanding and analysis:

Given the following grammar:

$$\begin{aligned} F &\rightarrow B\$F|B \\ B &\rightarrow D\#D|D \\ D &\rightarrow (F)|x|y \end{aligned}$$

where the terminals are $\$, \#, (,), x, y$ and F is the start non-terminal.

(a) (3 pt) Does \$ have a higher precedence than #? Justify your answer.

(b) (3 pt) Are # and \$ left or right associative? Justify your answer.

(c) (2 pt) Can the grammar generate the string $x\$x\#y\#(y\$x)$? If so, show the derivation; if not, justify your answer.

(d) (2 pt) Is this grammar ambiguous? Justify your answer.

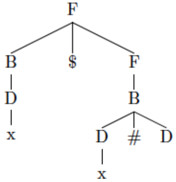
Sample Questions: (contd.)

Sol.

(a) No, # has a higher precedence because it is lower in the grammar rules, which means it will be evaluated first

(b) $ is right associative, and the associativity of # is decided by strings using parenthesis

(c) No, because the first y cannot be generated. See the following parse tree where it get stuck:



(d) (Hint:) The grammar is non-ambiguous. It contains two operators, $ and #. Here, $ is right-associative and, in case of the operator, # we can show that strings such as x#x#x cannot be generated by the grammar, this is restricted through the parenthesis. Show example string with parse tree for clarification ...

Strategy:

1. find features of strings: can be derived from the grammar.
2. prove by showing the strings donot have a feature.

AithLang

the ArithLang language

- Design decision: prefix, infix and postfix
- Contains only numbers and arithmetic operators

the Interpreter

- Reader: from a program to a AST, editing .g and automatically generate parser.java files
- Evaluator: traverse AST to generate values, 1) extending value types if needed 2) extending AST classes to store new nodes 3) update visitor class to compute values
- Printer

Sample Questions:

(b) (15 pt) Extend the language to support a "substring" operation on the positive numbers. The operation returns a section of numbers as well as the decimal point, if any, specified by an index range. If the range index is out of bound or when a non-positive integer is given, you return -1, indicating an error. See example scripts below:

```
$ (substr 50010 2 4)
$ 001
$ (substr 50010 1 1)
$ 5
$ (substr (let ((a 50010)) a) 2 4)
$ 001
$ (substr (+ 34 2) 1 (* 1 1 (- 4 2)))
$ 34
$ (substr -10 1 2)
$ -1
$ (substr 10 -1 2)
$ -1
$ (substr 10 1 4)
$ -1
$ (substr (/ 10 3) 0 4) # note that 10/3 = 3.333333...
$ 3.33
```

Sample Questions: (contd.)

Soln:

Assume the **SubStrExpr** is defined (so you do not need to write the definition of this AST node) and extended from **Exp** class with the following signature:

- constructor: `SubStrExpr(Exp src, Exp start, Exp end)`
- method: `public Exp getSrc()`
- method: `public Exp getStart()`
- method: `public Exp getEnd()`

As the first step, please modify the grammar below:

```
exp returns [Exp ast]:
| subs=substrexp { $ast = $subs.ast; }

substrexp returns [SubStrExpr ast]:
// complete grammar here
'(' 'substr'
    str=exp
    s=exp
    e=exp
')' { $ast = new SubStrExpr($str.ast, $s.ast, $e.ast); }
;
```

Sample Questions: (contd.)

Soln:

Then you can complete the following Evaluator:

```
@Override
public Value visit(SubStrExpr e, Env env) {
  // write the evaluation here
  NumVal value = (NumVal) e.getStr().accept(this, env);
  NumVal startValue = (NumVal) e.getStart().accept(this, env);
  NumVal endValue = (NumVal) e.getEnd().accept(this, env);

  if (value.v() >= 0) {
    String str = String.valueOf(value.v());
    int start = (int) startValue.v();
    int end = (int) endValue.v();

        if (start >= 0 && start <= end && str.length() > start && end <= str.
          ↪ length())
          return new StringVal(new String(str.substring(start, end)));
  }

  return new StringVal("-1");
}
```

VarLang and DefineLang

Variables

- Definition and use of a variable
- Scoping
- Free/bound variables
- Hole
- Environment: passing the *right* value to the expression; global variable will be defined in the initial environment and be visible entire interpreter life time

Sample Questions:

- (let ((x 10)) (let ((x 100)) (+x 1)))
  (let ((x 100)) (+x 1)) creates a hole for the definition (x 10) in the outer let expression

- (let ((a b)) a)
  F   B

4. (8 pt) Identify free and bound variables in the following expression. Write F (for free variables) or B (for bound variables) under each variables in the description.

```
(let ((foo (lambda (a b) (+ a c))) (c d) (d e)) (foo (+ a b c) (+ d e)))
                            B F       F     F    B      F F B     B F
```

FuncLang

Functions

- Lambda expression: lambda expression is a function, it has values, and can be passed as parameters, returns from a function and stored in the environment
- Call expression
- Function with a Name: Combine lambda expression with let and define expressions
- List and pair and their built-in functions for list: car, cdr, cons, null?
- Recursive calls
- High order functions
- Control structure, if then else
- Currying

FuncLang Programming

- Numbers: fibonnacci, summation, factorial
- List: **foldl**
- Simulate data structure: pair, tree
- **Sorting**
- **List and integer conversion**

Sample Questions:

5.4.2. Define a function `foldl` (fold left) with three parameters `op`, `zero_element` and `lst`. The parameter `op` itself is a two argument function, zero_element is the zero element of the operator function, e.g. 0 for plus function or 1 for multiply function, and lst is a list of elements. (plus function takes two parameters and adds them, multiply function takes two parameters and multiplies them.)

The function successively applies the `op` function to each element of the list and the result of previous `op` function (where no such results exists the zero element is used). The following interaction log illustrates foldl function

```
$ (define plus (lambda (x y) (+ x y)))
$ ( foldl  plus 0 ( list ))
0
$ ( foldl  plus 0 ( list  1))
1
$ ( foldl  plus 0 ( list  1 2))
3
$ ( foldl  plus 0 ( list  1 2 3))
6
$ ( foldl  plus 0 ( list  1 2 3 4))
10
```

foldl solution:

```
(define len (lambda (lst) (if (null? lst) 0 (+ 1 (len (cdr lst ))))))

(define foldl
    (lambda (op zero_element lst)
        (if (null? lst) 0
            (if (= 1 (len lst)) (op zero (car lst))
                (op (foldl op zero (cdr lst)) (car lst))
            )
        )
    )
)
```

sort solution:

```
(define max (lambda (lst) (if (null? lst) -1 (if (> (car lst) (max (cdr lst))) (car lst) (max (cdr lst)))))
)

(define rest (lambda (x l) (if (null? l) (list) (if (= (car l) x) (cdr l) (cons (car l) (rest x (cdr l)))))))

(define sort (lambda (lst) (if (null? lst) (list)
                                      (if (> (car lst) (max (cdr lst)))
                                          (cons (car lst) (sort (cdr lst)))
                                          (cons (max (cdr lst))
                                            (sort (rest (max (cdr lst)) lst)))))))
```

List to Integer conversion:

(b) (5 pt) Write a FuncLang program to compute an integer number from a list of digits from 0-9.

Example scripts:

$ ConvertInt(list (1 2 3))
$ 123
$ ConvertInt(list (0 2 3))
$ 23
$ ConvertInt(list (3 4 9 1))
$ 3491

Sol.

```
(define size (lambda (lst) (if (null? lst) 0 (+ 1 (size (cdr lst))))))
(define multp (lambda (n) (if (= n 1) 1 (if (> n 1) (* 10 (multp (- n 1))) 0))))
(define ConvertInt
  (lambda (lst)
    (let ((s (size lst)))
      (if (null? lst)
        0
        (+ (* (multp s) (car lst)) (ConvertInt (cdr lst)))))))
```