

# CS3723 Exam 1

Sep 25, 2009

1. (15pts) Finish the following sentences by filling in the blanks.
  - (a) Give two languages that belong to the functional programming paradigm:  
(1) Scheme; (2) Lisp; (3) ML.
  - (b) Give two languages that belong to the imperative programming paradigm:  
(1) C; (2) Fortran; (3) Pascal; (4) Perl;
  - (c) Give two languages that belong to the object-oriented programming paradigm: (1) Java; (2) C++; (3) Python.
  - (d) Give two advantages of using high-level programming languages such as C++/Java/Scheme: (1) easy to write and maintain; (2) easy to port to different machines.
  - (e) Give two advantages of using low-level assembly/machine programming languages: (1) direct access to hardware; (2) better machine efficiency.
  - (f) Give two advantages of using a compiled programming language: compiled code can run many times and error checking at compile time / can afford heavy weight optimizations;
  - (g) Give two advantages of using an interpreted programming language: have full knowledge of program input and can change program on the fly.
  - (h) Give two components that are required to implement both compilers and interpreters: (1) semantic analyzer; (2) syntax analyzer.
  - (i) Most programming languages are Turing complete and can express the class of partial-recursive functions. Give an example non-computable problem: The halting problem.
  - (j) The lexical syntax of languages can be formally expressed using Regular Expressions; The context-free syntax of languages can be formally expressed using BNF.
  - (k) The difference between concrete and abstract syntax is: Abstract Syntax is what compilers/interpreters internally use, and Concrete Syntax is what compilers/interpreters use to interact with programmers. Compilers/interpreters generally use Abstract Syntax Tree as a form of internal representation for input programs.
  - (l) Give two key differences between the functional programming style and the imperative programming style:
    - (1) functional programming treats functions as first class objects, while imperative languages do not;
    - (2) functional programming focuses on computing new values without modifying memory, while imperative programming focuses on modifying the values of existing variables.

- (m) Give two advantages of using the functional programming style (i.e., to build new values instead of modifying existing memory):
- (1) better parallelism, concurrency easier to detect;
  - (2) linked data structures can freely share data without worrying about unexpected side effects
- (n) A higher-order function is a function that takes other functions as parameters or returns other functions as results. A language is said to treat functions as first-class objects if it treats functions as primitive values. The difference between an expression and a statement is an expression has a value, while a statement does not.

2. (20pts) Language Syntax.

- (a) Suppose we have the following grammar,

$B ::= B < B \mid B \&\& B \mid N$

where  $N$  stands for all integer numbers.

- i. Give a parse tree for  $3 < 5 \&\& 6$ .

```

      B
     / | \
    B  && B
   / | \ |
  B < B 6
   |   |
   3   5

```

- ii. Give an AST (Abstract Syntax Tree) for  $3 < 5 \&\& 6$ .

```

      &&
     /  \
    <    6
   /  \
  3    5

```

- iii. Rewrite the grammar to be non-ambiguous so that both  $<$  and  $\&\&$  are left associative, and “ $<$ ” has higher precedence than “ $\&\&$ ”.

$B ::= B \&\& T \mid T$   
 $T ::= T < N \mid N$

NOTE: This question earns 4 points.

- (1pts) Well-formed BNF.
- (1pts) Associativity.
- (1pts) Precedence.
- (1pts) No ambiguity.

- (b) Give a context-free grammar (BNF) for each of the following languages. It is OK for your grammar to be ambiguous.

- i. All strings composed over the set of terminals  $\{0, 1, 2\}$ .

$\text{str} ::= \text{str char} \mid \text{char}$   
 $\text{char} ::= 0 \mid 1 \mid 2$

- ii. All strings composed over the set of terminals  $\{0, 1, 2\}$  that represent an even number (i.e., the number ends with 0 or 2).

```
even ::= str 0 | str 2 | 0 | 2
str  ::= str char | char
char ::= 0|1|2
```

- iii. All strings composed over the set of terminals  $\{0, 1, 2, (, )\}$  so that the left and right parentheses are properly nested and at least one number is inside each pair of  $()$ . For example,  $(21(02))$  and  $201(2(1))$  are in the language, but  $(( ))3$  or  $)3($  are not.

```
paren ::= ( paren ) | paren paren | num
num   ::= num digit | digit
digit ::= 0|1|2
```

(c) (15pts) Scheme Programming.

- i. Write a Scheme function *count* which takes an arbitrary value,  $x$ , as parameter and returns the number of atomic values contained in  $x$ . For example, invoking  $(\text{count } '(3\ 7\ 9\ z\ s\ (3\ (78\ 2\ 3))))$  should return 9, and invoking  $(\text{count } '(3\ (78\ s\ 3)))$  should return 4.

```
(define atom? (lambda (x) (or (symbol? x) (number? x) (boolean? x))))
```

```
(define count (lambda (x)
  (cond
    ((atom? x) 1)
    ((cons? x) (+ (count (car x))
                  (count (cdr x))))
    (else 0) ) ))
```

- ii. Write a Scheme function *countf* which takes two parameters: a function  $f$ , and an arbitrarily nested list,  $y$ . The *countf* function invokes  $f$  using each atomic value contained in  $y$  as parameter and returns the number of times that the invocation of  $f$  returns true. For example, invoking  $(\text{countf } (\text{lambda } (x) (\text{number? } x)) '(3\ 7\ 9\ z\ s\ (3\ (78\ 2\ 3))))$  should return 7, and invoking  $(\text{countf } (\text{lambda } (x) (\text{symbol? } x)) '(3\ 7\ 9\ z\ s\ (3\ (78\ 2\ 3))))$  should return 2.

```
(define countf (lambda (f x)
  (cond ((null? x) 0)
        ((cons? x) (+ (countf f (car x)) (countf f (cdr x))))
        (else (if (f x) 1 0))))
```