

CS3723 Midterm Exam 1

Feb 25, 2008

1. (30pts) Finish the following sentences by filling in the blanks.
 - (a) Programming languages can be implemented via two approaches: compilation and interpretation. Two advantages of the compilation approach are can afford heavy-weight optimizations and can detect errors early at compile time; two advantages of the interpretation approach are can change programs on the fly and are more knowledgeable about program behavior. Enumerate two components that are required to implement both compilers and interpreters: (1) syntax analysis, (2) semantic analysis.
 - (b) Most programming languages are equivalent in power and can express the class of partial recursive functions, which include all functions computable by a modern computer. An example of a non-computable problem is the halting problem. Lambda calculus is a small theoretical language but it is as powerful as practical languages such as C. The pure lambda calculus language supports a single type of value: function abstraction, and supports a single operation: function application. A lambda term is in normal form if it can not be further reduced. Lambda terms are said to be confluent because they either have a unique normal form or lead to infinite reductions.
 - (c) High-level languages such as C and Java have advantages over low-level machine languages because they are easier to maintain and are portable (machine independent). These languages can be separated into different programming paradigms. In particular, The C language belongs to the imperative paradigm, the Scheme language belongs to the functional paradigm, and the Java language belongs to the object-oriented paradigm. A key difference between Scheme and C is Scheme treat functions as first class objects, while C does not (other differences are also acceptable, e.g., Scheme program through expressions, while C through modifications and statements.).
 - (d) A higher-order function is a function that takes other functions as parameters or returns a function as result. A language is said to treat functions as first-class objects if it treats functions the same as primitive values. C does not treat functions as first-class objects because it does not allow functions to be returned as result.

- (e) The Lisp abstract machine includes five components: (1) The expression to evaluate, (2) the continuation (the rest of the program) (3) The Association list (variable-to-value map) (4) the cons cells, (5) garbage collector.
- (f) The lexical syntax of languages can be formally expressed using regular expressions; The context-free syntax of languages can be formally expressed using BNF. The difference between concrete and abstract syntax is concrete syntax is what programmers write to precisely express algorithms; abstract syntax is what compilers/interpreters use to internally represent the input program. Compilers/interpreters generally use AST (Abstract Syntax Tree) as a form of internal representation for input programs.

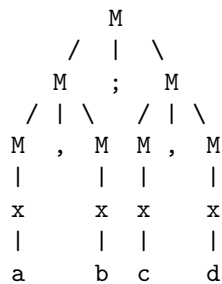
2. (20pts) Language Syntax.

- (a) Suppose we have the following grammar,

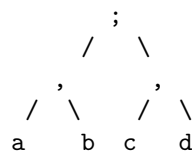
$M ::= x \mid M, M \mid M ; M$

where x stands for all variable names. Give a parse tree and an abstract syntax tree for the expression $a, b; c, d$. Rewrite the grammar to be non-ambiguous so that “;” is left associative, “,” is right associative, and “;” has higher precedence than “,”.

The parse tree:



The AST:



The rewritten grammar.

$M ::= M ; N \mid N$
 $N ::= x , N \mid x$

- (b) Give the context-free grammar for a small language that expresses the coordinates of a collection of objects. In particular, each sentence of the language is a sequence of objects separated by “,” and

enclosed inside a pair of $\{ \}$, and each object is composed of a name followed by a pair of coordinates in the form of $(i1, i2)$, where $i1$ and $i2$ are arbitrary integers. The terminals of the language includes SYM, which represents all names of objects; INT, which represents all integer numbers; and $\{ \}$ (,). For example, $\{ F(1,2), N(3,-5) \}$ and $\{ \text{point}(-1,0) \}$ are both valid sentences in the language, but $\{ F(2) \}$ (missing the second coordinate) and $\{ \text{point}(-1,0) Q(0,2) \}$ (missing “,” between the two objects) are not. Using your grammar, give a parse tree and AST for $\{ F(1,2), N(3,-5) \}$.

The grammar:

```
L ::= { objects }
objects ::= objects , obj | obj
obj ::= SYM ( INT , INT)
```

The parse tree:

```

      L
     / | \
    {  |  }
    {  |  }
    {  |  }
    objects , obj
      |   / | | \ \
      obj SYM(INT,INT)
        /| | | \ \ | | \
        SYM(INT,INT) N 3 -5
          | | |
          F 1 2
```

AST:

```

      {}
      |
      ,
     / \
    obj obj
   / | \ / | \
   F 1 2 N 3 -5
```

3. (20pts) Reduce each lambda term to normal form.

- (a) $(\lambda x. \lambda y. x y y) (\lambda z. z)$
 $\Rightarrow \lambda y. (\lambda z. z) y y$
 $\Rightarrow \lambda y. y y$
- (b) $(\lambda y. (\lambda x. \lambda y. x (x y)) (\lambda g. g y)) 5$
 $\Rightarrow (\lambda x. \lambda y. x (x y)) (\lambda g. g 5)$
 $\Rightarrow \lambda y. (\lambda g. g 5) ((\lambda g. g 5) y)$
 $\Rightarrow \lambda y. (\lambda g. g 5) (y 5)$
 $\Rightarrow \lambda y. (y 5) 5$

4. (20pts) Scheme Programming.

- (a) Write a Scheme function *count* which takes two parameters: *x*, an atomic value; and *y*, an arbitrary expression. The function returns how many times that *x* has appeared in *y*. For example, invoking `(count 3 '(3 7 9 z s (3 (78 2 3))))` should return 3, and invoking `(count 's '(9 z s (3 (78 s 3))))` should return 2.

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

- (b) Write a Scheme function *collect_numbers* which takes an arbitrary expression, *y*, and returns an expression that contains only the numbers in *y* (i.e., all the other elements in *y* are removed). For example, invoking `(collect_numbers '(lambda x x x y 3 7) 9 (x z))` should return a list `'((3 7) 9)`.

```
(define collect_numbers
  (lambda (y)
    (cond ((number? y) y)
          ((cons? y)
           (let ((first (collect_numbers (car y)))
                 (second (collect_numbers (cdr y))))
             (if (null? first) second (cons first second))))
          (else '()))))
```

5. (10pts) Language translation. Translate the following C program to lambda calculus.

```
int foo(int x, int y) { return x + 2 * y; }
int map(f, x1, x2) { return f (f(x1,x2), f(x2, x1)); }
int main()
{
  return map (foo, 2, 3);
}
```

$(\lambda f. \lambda x1. \lambda x2. f (f x1 x2) (f x2 x1)) (\lambda x. \lambda y. x + 2 * y) 2 3$