

# Midterm Exam

CSC 254

17 October 2006

## Directions; PLEASE READ

This exam comprises a mixture of short-answer and problem-solving questions. Values are indicated for each; they total 100 points. Questions 5(f) and 7 are worth 10 and 8 extra credit points, respectively; they won't factor into your exam score, but may help to raise your letter grade at the end of the semester.

This is a *closed-book* exam. You must put away all books and notes (except for a dictionary, if you want one). Please confine your answers to the space provided.

Put your name on every page. That way if I lose a staple I won't lose your answers. (Please do this **NOW** so you don't forget.) Scratch paper is available if you need it, but the proctor will collect **only the exams**.

In the interest of fairness, the proctor has been instructed not to answer questions during the exam. If you are unsure what a question is asking, make a reasonable assumption and state it as part of your answer.

You must complete the exam in class. The proctor will collect any remaining exams promptly at 4:40 pm. Good luck!

1. (8 points) Compilation is typically divided into a series of *phases*, each of which transforms its input in a well-defined way. One or more consecutive phases constitute a *pass* of compilation if they don't start until after all previous phases have completed, and they end before any subsequent phases begin. Explain the rationale for passes: why might we want to structure phases this way?

**Answer:** Possible answers include:

- If the front end and back end of the compiler are separate passes, then the same front end can easily be paired with back ends for several target machines, and the same back end can easily be paired with front ends for several different languages.
  - If passes are structured as separate programs, then they need not reside in memory simultaneously. Separation of passes may therefore allow a large compiler to run on a machine with a very small amount of main memory.
  - If the phase B needs random or non-left-to-right access to the output of the phase A, we may need to finish phase A before beginning phase B.
  - If there is a fatal error in the front end and we discover it before starting the back end, we may never need to run the back end at all, since it isn't supposed to produce any error messages, and we won't be using its output.
2. (12 points) Explain why an integer variable, local to a subroutine, might be allocated in static storage in Fortran 77, in the stack in C, and in the heap in Lisp.

**Answer:** Fortran 77 lacks recursion, so there can never be more than one instance of a given local variable at a time. C has recursion, but specifies that the lifetime of a (non-static) local variable ends when the subroutine returns. Lisp has unlimited extent: the lifetime of a local variable ends when no valid references to it remain. Since such references may outlive a subroutine instance, space cannot necessarily be reclaimed when the subroutine returns.

3. (9 points) Briefly describe three ways in which scripting languages differ from traditional imperative languages.

**Answer:** Possibilities include: both batch and interactive use; economy of expression; lack of declarations, simple scoping rules; flexible dynamic typing; easy access to other programs; sophisticated pattern matching and string manipulation; and high-level data types. For further details see Section 13.1.1 in the text.

4. (10 points) How does one tell whether a grammar is LL(1)?

**Answer:** Look for overlapping PREDICT sets. More precisely, G is LL(1) if and only if  $\forall$  nonterminals  $X$  and all productions  $X \rightarrow \alpha$  and  $X \rightarrow \beta$ ,  $\text{PREDICT}(X \rightarrow \alpha) \cap \text{PREDICT}(X \rightarrow \beta) = \emptyset$ .

5. Consider the following context-free grammar.

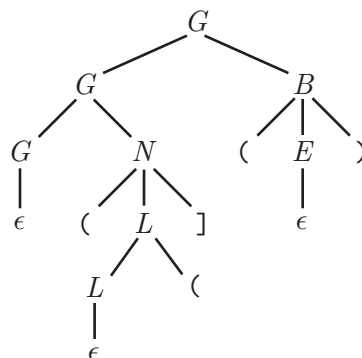
$$\begin{aligned} G &\rightarrow G B \\ &\rightarrow G N \\ &\rightarrow \epsilon \\ B &\rightarrow ( E ) \\ E &\rightarrow E ( E ) \\ &\rightarrow \epsilon \\ N &\rightarrow ( L ] \\ L &\rightarrow L E \\ L &\rightarrow L ( \\ &\rightarrow \epsilon \end{aligned}$$

- (a) (6 points) Describe, in English, the language generated by this grammar (Hint:  $B$  stands for “balanced”;  $N$  stands for “nonbalanced”). Please do *not* give me a low-level description of how the grammar works. I’m looking for a high-level characterization of the language—one that is independent of the particular grammar chosen.

**Answer:** Strings of balanced parentheses, with the exception that a right square bracket matches all still-open left parentheses that precede it. (Some dialects of Lisp permit these “close all” brackets.)

- (b) (9 points) Give a parse tree for the string  $((])$ .

**Answer:**



- (c) (8 points) Give a rightmost (canonical) derivation of this same string.

**Answer:**  $G$   
 $G \ B$   
 $G \ ( \ E \ )$   
 $G \ ( \ )$   
 $G \ N \ ( \ )$   
 $G \ ( \ L \ ] \ ( \ )$   
 $G \ ( \ L \ ( \ ] \ ( \ )$   
 $G \ ( \ ( \ ] \ ( \ )$   
 $( \ ( \ ] \ ( \ )$

- (d) Recall that FIRST and FOLLOW sets are defined for symbols in an arbitrary context-free grammar (regardless of parsing algorithm).

- i. (5 points) What is FIRST( $E$ ) in our context-free grammar?

**Answer:**  $\{ ( \}$  or  $\{ (, \epsilon \}$ , depending on whether you use the definition of FIRST from lecture or from the text (I'll accept either).

- ii. (5 points) What is FOLLOW( $E$ )?

**Answer:**  $\{ (, ), ] \}$ .

- (e) (10 points) Add attribute rules to the context-free grammar to create an attribute grammar that calculates, for every right square bracket ( $]$ ) an attribute  $n$  that indicates the number of left parentheses to the left of the bracket that do not yet have a matching right parenthesis. For example, in the string  $((] ($ ), the bracket's  $n$  would be 2. (Note: in lecture the attributes of tokens were always synthesized, but there is no reason we can't have inherited attributes, too.)

**Answer:** Note that  $B$  and  $E$  are irrelevant to  $]n$ .

$G \rightarrow G B$   
 $G \rightarrow G N$   
 $G \rightarrow \epsilon$   
 $B \rightarrow ( E )$   
 $E \rightarrow E ( E )$   
 $B \rightarrow \epsilon$   
 $N \rightarrow ( L ] \quad \triangleright \ ].n := L.n + 1$   
 $L \rightarrow L E \quad \triangleright \ L_1.n := L_2.n$   
 $L \rightarrow L ( \quad \triangleright \ L_1.n := L_2.n + 1$   
 $L \rightarrow \epsilon \quad \triangleright \ L.n := 0$

- (f) (Extra credit; up to 10 points) Given its use of left recursion, our grammar is clearly not LL(1). Does this language have an LL(1) grammar? Explain.

**Answer:** No it doesn't. Proving this is difficult, but the intuition is straightforward: In an LL(1) parser, the contents of the parse stack represents the predicted remainder of the program, and decisions are always made on the basis of the top-of-stack symbol and the next token of input. When it matches a left parenthesis, the parser must predict whether it will see a matching right parenthesis, or whether this will be "swallowed up" by a right square bracket. Since the distance to the right parenthesis or bracket is unbounded, the parser cannot make this prediction with a fixed sized grammar. Note that eliminating left recursion does *not* suffice to make the grammar LL(1): there is potentially unbounded overlap between prefixes of the yields of  $B$  and  $N$ .

6. Consider the following program in Scheme:

```
(define A
  (lambda ()
    (let* ((x 2)
           (C (lambda (P)
                 (let ((x 4))
                   (P))))
           (D (lambda ()
                 x))
           (B (lambda ()
                 (let ((x 3))
                   (C D))))))
      (B))))
```

*Explain your answer* for each of the following sub-problems.

(a) (6 points) What does this program print?

**Answer:** 2. Scheme uses static scope, so D sees the `x` in A.

(b) (6 points) What would the program print if Scheme used dynamic scope and shallow binding?

**Answer:** 4. D is eventually called from C, so the most recently declared `x` is the one in C.

(c) (6 points) What would the program print if Scheme used dynamic scope and deep binding?

**Answer:** 3. D is bound to a referencing environment in B, when it is passed to C. At that point the most recently declared `x` is the one in B.

7. (Extra credit; up to 8 points) We learned in class that unlimited extent may be required in Scheme if we return a function from some other function:

```
(define plus-k
  (lambda (k)
    (lambda (n) (+ n k))))
(let* ((k 1)
       (foo (plus-k 2)))
  (foo 3))                ==> 5
```

How would this code behave if Scheme had dynamic scope and shallow binding? Would the language still need unlimited extent?

**Answer:** If Scheme had dynamic scope and shallow binding, function `plus-k` would ignore its argument. When `foo` was called it would use the referencing environment of the `let*`, in which `k` is 1, and the expression would evaluate to 4. Unlimited extent would not matter in this case. It would still matter in other programs, however:

```
(define foo
  (lambda()
    (let ((X '(a b c))
          (Y '(d e f)))
      (cons X Y))))
(cadr (foo))           ==> d
```

Here `foo` returns a reference to a cons cell containing references to `X` and `Y`, which must continue to exist.