Midterm Exam

CSC 2/454

18 October 2012

Directions; PLEASE READ

This exam comprises a mixture of multiple-choice, short-answer and problem-solving questions. Values are indicated for each; they total 72 points. Question 17 is worth 8 extra credit points. It's not part of the 72, and it won't factor into your exam score, but it 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, cellphones, and notes. Please confine your answers to the space provided. For multiple choice questions, unless otherwise instructed, darken the circle next to the single best answer. Be sure to read all candidate answers before choosing. No partial credit will be given on the multiple-choice questions.

In the interest of fairness, I will decline 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. I will collect any remaining exams promptly at 1:45 pm. Good luck!

1. (3 points) Put your name on every page (so if I lose a staple I won't lose your answers).

Multiple Choice

2. (2 points) Which of the following languages is the newest?

⊗ **a.** C
◯ **b.** Fortran
◯ **c.** Lisp
◯ **d.** Simula

3. (2 points) Which of the following languages is the oldest?

⊗ **a.** Perl
◯ **b.** PHP
◯ **c.** Python
◯ **d.** Ruby

4. (2 points) Which of the following is typically *not* the job of the scanner?

◯ **a.** remove comments
⊗ **b.** expand macros
◯ **c.** tag tokens with line numbers
◯ **d.** save text of identifiers, numbers, strings, etc.

5. (2 points) What is the principal motivation for syntax error recovery?

◯ **a.** to allow the compiler to output a better error message

◯ **b.** to figure out what the program should have said, so code generation knows what to produce

⊗ **c.** to allow the compiler to keep looking for additional errors

◯ **d.** none of the above

6. (2 points) The static chain

◯ **a.** allows a running program to find variables whose lifetime spans the entire program execution.

◯ **b.** identifies objects imported into the current scope.

⊗ **c.** allows a subroutine to find the frame of the lexically surrounding routine.

◯ **d.** identifies all modules that were linked together in a single object file.

7. (2 points) A higher-order function is

⊗ **a.** one that takes other functions as parameters, or returns a function as a result.

◯ **b.** one at the outermost level of lexical nesting.

◯ **c.** one whose local variables have unlimited extent.

◯ **d.** one that has been encapsulated with its referencing environment in a closure.

8. (4 points) Which of the following correctly characterize a tradeoff between overloading and parametric polymorphism? **Check as many as apply**; this will be graded as if it were 4 true–false questions.

◯ **a.** Overloading is implemented as multiple separate functions in the generated code; parametric polymorphism is implemented as a single function that can inspect the types of its arguments.

⊗ **b.** The programmer *thinks* of overloading as multiple things with one name; he or she thinks of parametric polymorphism as one thing with multiple behaviors, or with a general-purpose behavior applicable to multiple types.

◯ **c.** Overloading is resolved at compile time; parametric polymorphism is resolved dynamically.

⊗ **d.** Overloading requires the programmer to enumerate the types for which an operation is supported; parametric polymorphism allows the programmer to specify a function that can apply to a potentially unbounded set of types.

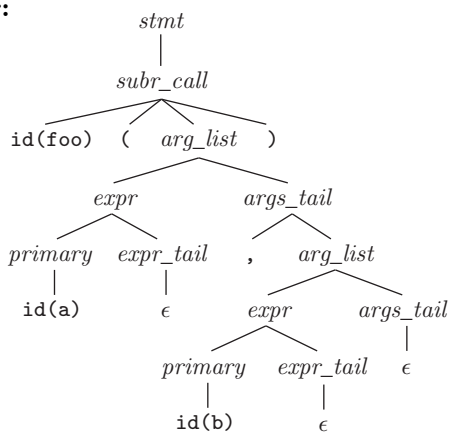
Questions 9 through 13 refer to the following context-free grammar.

stmt → *assignment*
→ *subr_call*
assignment → `id :=` *expr*
subr_call → `id (` *arg_list* `)`
expr → *primary* *expr_tail*
expr_tail → *op* *expr*
→ ϵ
primary → `id`
→ *subr_call*
→ `(` *expr* `)`
op → `+`
→ `-`
→ `*`
→ `/`
arg_list → *expr* *args_tail*
args_tail → `,` *arg_list*
→ ϵ

Figure 1: A simple CFG.

9. (6 points) Using the grammar of Figure 1, construct a parse tree for the input string `foo(a, b)`.

Answer:



10. (6 points) Give a canonical (rightmost) derivation of this same string. (Hint: the answer is 13 lines long.)

Answer:

stmt
subr_call
`id (` *arg_list* `)`
`id (` *expr args_tail* `)`
`id (` *expr* `,` *arg_list* `)`
`id (` *expr* `,` *expr args_tail* `)`
`id (` *expr* `,` *expr* `)`
`id (` *expr* `,` *primary expr_tail* `)`
`id (` *expr* `,` *primary* `)`
`id (` *expr* `, id )`
`id (` *primary expr_tail* `, id )`
`id (` *primary* `, id )`
`id ( id , id )`

11. (6 points) Prove that the grammar of Figure 1 is not LL(1).

    Answer: The token `id` is in both FIRST(*assignment*) and FIRST(*subr_call*), and thus in both PREDICT(*stmt* → *assignment*) and PREDICT(*stmt* → *subr_call*). Likewise, `id` is in both FIRST(`id`) and FIRST(*subr_call*), and thus in both PREDICT(*primary* → `id`) and PREDICT(*primary* → *subr_call*). Since the predict sets for both *stmt* and *primary* are not disjoint, the grammar is not LL(1).

12. (6 points) Modify the grammar so it *is* LL(1).

    Answer: Eliminate the *subr_call* production and replace the *stmt* and *primary* productions with the following (all else remains the same).

    stmt → `id` *stmt_tail*
    stmt_tail → `(` *arg_list* `)`
    → `:=` *expr*
    primary → `id` *primary_tail*
    → `(` *expr* `)`
    primary_tail → `(` *arg_list* `)`
    → ϵ

13. (6 points) Flip back to Figure 1 and add attribute rules to the grammar to accumulate into the root of the tree a count of the maximum depth to which parentheses are nested in any expression in the input statement. For example, given

    ```
    f1(a, f2(b * (c + (d - (e - f)))))
    ```

    the *stmt* at the root of the tree should have an attribute with a count of 3 (note: the parentheses surrounding argument lists don't count).

Answer:

stmt → *assignment*	▷	stmt.depth = assignment.depth
stmt → *subr_call*	▷	stmt.depth = subr_call.depth
assignment → `id :=` *expr*	▷	assignment.depth = expr.depth
subr_call → `id (` *arg_list* `)`	▷	subr_call.depth = arg_list.depth
expr → *primary expr_tail*	▷	expr.depth = max(primary.depth, expr_tail.depth)
expr_tail → *op expr*	▷	expr_tail.depth = expr.depth
expr_tail → ϵ	▷	expr_tail.depth = 0
primary → `id`	▷	primary.depth = 0
primary → *subr_call*	▷	primary.depth = subr_call.depth
primary → `(` *expr* `)`	▷	primary.depth = expr.depth + 1
op → `+`		
op → `-`		
op → `*`		
op → `/`		
arg_list → *expr args_tail*	▷	args_list.depth = max(expr.depth, args_tail.depth)
args_tail → `,` *arg_list*	▷	args_tail.depth = args_list.depth
args_tail → ϵ	▷	args_tail.depth = 0

14. (8 points) Using any languages you want as examples, list two language rules (restrictions the programmer must follow) that you would expect to be enforced by the scanner, two that you would expect to be enforced by the parser, two that you would expect to be enforced by the semantic analyzer, and two that you would expect to be enforced at run time.

    Answer: Here are some examples for Java:

    scanner Certain characters (e.g. the @ sign and the back-quote) are allowed only within comments and strings. A keyword (`while`, `if`, `class`, etc.) must be followed by white space to separate it from any subsequent token beginning with a letter or digit. `/* */` comments cannot be nested.

    parser Parentheses and `{...}` pairs must be balanced. Binary operators must appear between a pair of operands. Statements must be followed by a semicolon; function definitions must not. In general, the program must follow the syntax specified by a Java CFG.

    semantic analyzer Function declarations and calls must have matching numbers and types of arguments. The left and right-hand sides of an assignment statement must have compatible abstract types. A `break` statement cannot appear except inside a loop or `switch` statement. The labels on the arms of a `switch` statement must be constants, and must be distinct.

    run time Array subscripts must be within bounds. The concrete type of the right-hand side of an assignment statement must be compatible with the abstract type of the left-hand side (no "backward" assignment). The denominator of a division operation must not be zero.

15. (9 points) Consider the following function in Haskell:

```
f = let x   = 2
        c p = let x = 4 in p
        d   = x
        b   = let x = 3 in c d
    in b
```

Explain your answer for each of the following sub-problems. Hint: for purposes of this question, you can safely think of `f`, `d`, and `b` as zero-argument functions.

(a) What will function `f` return when called?

Answer: 2. Haskell uses static scope, do `d` sees the `x` in `f`.

(b) What would `f` return if Haskell had dynamic scope and shallow binding?

Answer: 4. When `d` is eventually called from `c`, the most recently elaborated `x` is the one in `c`.

(c) What would `f` return if Haskell had dynamic scope and deep binding?

Answer: 3. The referencing environment for `d` is bound in `b`, when `d` is passed to `c`. At that point the most recently elaborated `x` is the one in `b`.

16. (6 points) Using any languages you want as examples, list two kinds of objects (in an informal sense of the word) that are generally allocated statically, two that are generally allocated in the stack, and two that are generally allocated in the heap.

Answer:

static code; global variables; constants; `own` (`static`) variables; local variables in Fortran; compiler-generated tables for debugging, exception handling, garbage collection, etc.

stack local variables and parameters in most languages; saved registers; return addresses; temporary values (spilled registers)

heap variables allocated via `new` in C++ or Java; dynamically-resizable strings or arrays; local variables with unlimited extent

17. (Extra Credit; 8 points max) Summarize the philosophical debate between languages with lots of early binding (e.g., C) and languages with lots of late binding (e.g., Perl). Which do you prefer? Why?

Answer: Languages with lots of early binding tend to produce more efficient programs, because they avoid the need for run-time checks. They also tend to catch errors sooner, which may be good for reliability and software maintenance.

Languages with lots of late binding tend to make it easier to write programs quickly and to write functions that can be used in multiple ways (e.g., for different argument types).

For further discussion, see the sidebar on p. 717 of the text.