Midterm Exam

CSC 254

18 October 2005

Directions; PLEASE READ

This exam comprises a mixture of short-answer and problem-solving questions. Values are indicated for each; they total 90 points. Question 6(e) is worth 8 extra credit points; it 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. 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. (7 points) List, in order, the principal phases of compilation.

   Answer: Scanning, parsing, semantic analysis, intermediate code generation, machine-independent code improvement, target code generation, machine-dependent code improvement.

2. (8 points) State the circumstances under which a good compiler will use linear search, binary search, hashing, or a characteristic array to implement a `case` (`switch`) statement.

   Answer: If the number of labels is small, use linear search. Else if there are any big ranges, use binary search. Else if the labels form a dense set, use a characteristic array. Else use hashing.

3. (6 points) Is it possible to write a tail-recursive version of the classic quicksort algorithm? Why or why not?

   Answer: Not without continuation-based surgery that would render it unrecognizable. Quicksort depends fundamentally on making *two* recursive calls and returning the concatenation of their results.

4. (6 points) Explain the fundamental distinction between enumeration-based and condition-based iteration.

Answer: In enumeration-based synchronization the loop is executed once for each member of a finite set, so the number of iterations is known in advance. In condition-based synchronization the loop is executed until its side effects make some Boolean condition true.

5. Consider the following program in C#:

```
using System;
public delegate int UnaryOp(int n);
   // type declaration: UnaryOp is a function from ints to ints

public class Foo {
    static int a = 2;
    static UnaryOp b(int c) {
        int d = a + c;
        Console.WriteLine(d);
        return delegate(int n) { return c + n; };
    }
    public static void Main(string[] args) {
        Console.WriteLine(b(3)(4));
    }
}
```

The keyword `delegate` in C# has roughly the meaning of `lambda` in Scheme.

(a) (5 points) What does this program print?

Answer:

```
5
7
```

(b) (8 points) Consider objects `a`, `b`, `c`, and `d` (in the informal sense of the word "object"). Which of these, if any, is likely to be statically allocated? Which could be allocated on the stack? Which would need to be allocated in the heap? Explain.

Answer: Objects `a` and `b` are globally defined, and could be statically allocated. (The code for the anonymous function returned by `b` could likewise be statically allocated.) Local variable `d` could be allocated on the stack. While parameters can often be allocated on the stack, `c` has unlimited extent (it is needed by the anonymous function returned by `b`), so it has to go in the heap.

6. Consider the following grammar for variable declarations in some Algol-family language:

decs → `VAR` *dec more_decs* `$$`

dec → *id_list* `:` `ID` `;`

more_decs → *dec more_decs*

→

id_list → `ID` *more_ids*

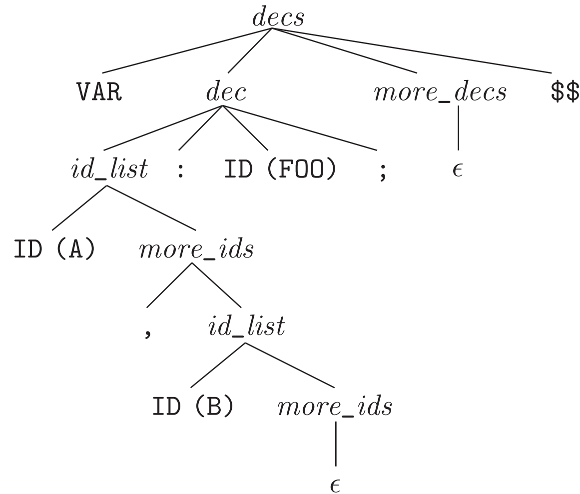
more_ids → `,` *id_list*

→

The `ID` after the colon in the second production is meant to be a type name.

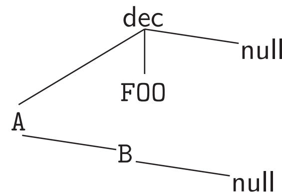
(a) (10 points) Give a parse tree for the string `VAR A , B : FOO ; $$`.

Answer:



(b) (6 points) Give a plausible syntax tree for the same string (you'll have to invent a reasonable structure).

Answer:



(c) (10 points) Show a trace of the stack of an LL(1) parser on this input. On each line of the trace, show the complete contents of the stack and the remaining input. Start a new line each time a symbol is matched or a production predicted.

Answer:

Stack	Remaining input
decs	`VAR A, B : FOO; $$`
`VAR` dec more_decs	`VAR A, B : FOO; $$`
dec more_decs	`A, B : FOO; $$`
id_list `:` `ID` `;` more_decs	`A, B : FOO; $$`
`ID` more_ids `:` `ID` `;` more_decs	`A, B : FOO; $$`
more_ids `:` `ID` `;` more_decs	`, B : FOO; $$`
`,` id_list `:` `ID` `;` more_decs	`, B : FOO; $$`
id_list `:` `ID` `;` more_decs	`B : FOO; $$`
`ID` more_ids `:` `ID` `;` more_decs	`B : FOO; $$`
more_ids `:` `ID` `;` more_decs	`: FOO; $$`
`:` `ID` `;` more_decs	`: FOO; $$`
`ID` `;` more_decs	`FOO; $$`
`;` more_decs	`; $$`
more_decs	`$$`

(d) (10 points) Add action routines to the grammar, to insert the declared variables into the symbol table by calling the routine void declare_variable(name new_var, name type) at appropriate times during the parse. You may assume that `ID` has a synthetic attribute `n` of type name that is provided by the scanner. You may invent any other attributes

you need to support your action routines (but of course you have to give the code to fill them in). It must of course be possible to execute your action routines in the course of a top-down parse. Hint: at least one of your attributes will need to hold a list.

Answer: Give *id_list* and *more_ids* an attribute nl of type list<name>. Then

decs → VAR *dec more_decs*

dec → *id_list* : ID ;
{ foreach v in id_list.nl declare_variable(v, ID.n) }

more_decs → *dec more_decs*

→

id_list → ID *more_ids* { id_list.nl = cons(ID.n, more_ids.nl) }

more_ids → , *id_list* { more_ids.nl = id_list.nl }

→ { more_ids.nl = null }

(e) (**Extra Credit**; up to 8 points) Write an attribute grammar that accomplishes the same result as the previous subproblem, but without any attribute that contains a list or other value of unbounded size. Hint: your attribute grammar won't be complicated, but it won't be L-attributed either.

Answer: Give *id_list* and *more_ids* an inherited attribute t of type name. Then

decs → VAR *dec more_decs*

dec → *id_list* : ID ;
▷ id_list.t = ID.n

more_decs → *dec more_decs*

→

id_list → ID *more_ids*
▷ declare_variable (ID.n, id_list.t)
▷ more_ids.t = id_list.t

more_ids → , *id_list*
▷ id_list.t = more_ids.t

→

7. (6 points) Consider the following pseudocode:

```
x : integer        -- global

procedure set_x(n : integer)
    x := n

procedure print_x
    write_integer(x)

procedure first
    set_x(1)
    print_x
```

```
procedure second
    x : integer
    set_x(2)
    print_x

set_x(0)
first()
print_x
second()
print_x
```

What does this program print if the language uses static scoping? What does it print with dynamic scoping? Why?

Answer: With static scoping it prints `1 1 2 2`. With dynamic scoping it prints `1 1 2 1`. The difference lies in whether set_x sees the global x or the x declared in second when it is called from second.

8. (8 points) Consider the following pseudocode:

```
x : integer    -- global

procedure set_x(n : integer)
    x := n

procedure print_x
    write_integer(x)

procedure foo(S, P : function; n : integer)
    x : integer := 5
    if n in {1, 3}
        set_x(n)
    else
        S(n)
    if n in {1, 2}
        print_x
    else
        P

set_x(0); foo(set_x, print_x, 1); print_x
set_x(0); foo(set_x, print_x, 2); print_x
set_x(0); foo(set_x, print_x, 3); print_x
set_x(0); foo(set_x, print_x, 4); print_x
```

Assume that the language uses dynamic scoping. What does the program print if the language uses shallow binding? What does it print with deep binding? Why?

Answer: With shallow binding, `set_x` and `print_x` always access `foo`'s local `x`. The program prints

```
1 0  2 0  3 0  4 0
```

With deep binding, `set_x` accesses the global `x` when `n` is even and `foo`'s local `x` when `n` is odd. Similarly, `print_x` accesses the global `x` when `n` is 3 or 4 and `foo`'s local `x` when `n` is 1 or 2. The program prints

```
1 0  5 2  0 0  4 4
```