NAME_____GRADE_____

Prob	1	2	3	4	Σ	
Max	25	25	25	25	100	
Score						

CSc520 **midterm exam** 27 March 2006

DUE Monday 3 April in class

Instructions: Do all the problems.

Submit your solutions using a typesetting program such as *TeX* or *troff*, that is capable of handling appropriate type faces and equations. Start each solution on a *new page*, identify the problem clearly, and number each page. Use *one side* of the paper only. Submit answers in the envelope provided.

Clarity and conciseness will earn points, so revise your work before submission, and consider how your work is presented.

Some problems require you to describe syntax or semantics of a language. Your syntactic/semantic descriptions should include thorough comments to explain the ideas behind each specification in clear technical English; the existence of a formalism does not relieve you of this responsibility, any more than it does in writing code. When precise semantics is called for, divide your solution into (1) syntax changes (2) semantic domain changes (3) semantic function changes. Under (1), describe any *contextual constraints* (static semantics) informally—there is no need to formalize these. Under (3), give signatures for all semantic functions and auxiliary functions introduced.

Important: You are reminded of the provisions regarding academic dishonesty in the Code of Academic Integrity, and in the syllabus of this course. Work in this course is to be done by individuals only, not in groups. The sharing of answers from another student in this course, or from another student with notes from earlier versions of this course, is prohibited. *All work is expected to be that of each student alone, without consultation with others*, and not the product of team efforts or collaboration with other authors. Plagiarism or the incorporation of another person's words or ideas constitutes theft of intellectual property, and will be dealt with as such. Use of a current or previous student's notes, solutions or materials is a prohibited activity.

1. Pointers in IMP

We want to add pointers to the imperative language IMP (Example 3.6, p. 66 of the Watt text, called IMP_0 in class). The new language will be called IMP_p. To this end, we change the syntax of the language as follows:

1. The following productions for `Expression` are added to allow the creation and use of pointers. The behavior of these constructs is as in C:

```
Expression ::=   . . .
      | & Identifier
      | * Identifier
```

2. The following production for `Command` is added to allow assignments through pointers:

```
Command ::=  . . .
      | *Identifier := Expression
```

3. The syntax and semantics for declarations must be changed to allow the declaration of pointers—this is part of the problem posed below.

Pointers follow the following rules and restrictions:

(i) A pointer can be stored in a location. Thus, the following program is legal (assuming the appropriate declarations), and should assign to the location of `w` the value 2.

```
let ... in
     x := &y ;
    *x := 1  ;
     w := y + *x ;
```

(ii) Only expressions that denote locations may be assigned to. Thus, the following is illegal (you need to consider only legal programs):

```
let const x ~ 10 in *x := 1 ;
```

(iii) The only pointer arithmetic that is permitted is the adding of an offset to a pointer value. The following is legal, assuming `p` is of type ''pointer to integer'' and `y` is of type ''integer''.

```
let ... in p := &y + 3 ;
```

(a) Provide any further extensions of the language syntax needed, including the extended syntax of declarations and expressions. Allowance must be made for pointers to integers as well as pointers to booleans.

(b) Give all required semantic domains for this language, and explain how such domains have been changed from IMP_0 to accommodate pointers in IMP_p.

(c) Provide signatures for all semantic functions, and explain how they have changed from those for IMP_0.

(d) Provide all needed semantics for this extension of IMP_0. Give only the extensions that are *necessary* to accommodate the new syntax.

2. Denotational Semantics of `case`

Add the following **`case`** control structure to the language IMP (use the simplest IMP, called IMP_0 in class and found in Watt, Example 3.6)

The command **`case`** E **`of`** N_1 `:` C_1 `;` $\cdots$ `;` N_n `:` C_n **`end`** has the effect of executing whichever subcommand C_i is labeled by a numeral N_i whose value matches the (integer) value of E. In the absence of such a match, the **`case`** command executes a **`skip`**.

(a) Describe an abstract syntax for this extension

```
Command ::=  ... |
```

(b) What are the contextual constraints, if any?

(c) Give the semantic function or functions for handling this construct. Your semantic functions must use recursion, and should *not* have an ellipsis ($\cdots$) anywhere in them. (See comments on this subject in Homework 4).

(For simplicity, do *not* extend the language to accommodate a **`break`** command.)

(d) Describe in words what your semantics implies is the meaning of a **`case`** when the value of E matches more than one of the N_i values. Your semantics should always give an unambiguous result, and different reasonable possibilities exist. State what *your* semantics in (c) does, and explain why.

(e) In some programming languages, such as **`C`**, after the code for one of the cases is executed, control ''falls through'' to the following case and continues execution. In other languages, such as ADA, execution resumes at the end of the **`case`** command, and skips all other cases. Describe in words what your semantics does, and explain how your semantic functions achieve your intended result.

(f) Briefly, discuss how the semantics of **`case`** would be affected if side-effects were allowed in expression evaluation. (Do *not* give a full semantics, but describe the main difference in the semantics from (c) in a non-formal but clear and precise way).

3. Indexed Loops in IMP

Most languages have an "indexed loop" command with a syntax similar to

for *j* **:** *first*, *last*, *step* **do** *comm* **end**

where *j* is an identifier, *first*, *last*, *step* are integer-valued expressions and *comm* is an command (which may involve *j*). Languages differ considerably in the semantics of indexed loops. For example:

- In most languages, the expressions *first*, *last*, *step* are considered to be evaluated *once* upon loop initialization, so that the number of iterations of the body can be determined at loop entry from the initial values *F*, *L* and *S* of these expressions. Thus these values *F*, *L*, *S* do not change even if the expressions *first*, *last*, *step* involve variables changed by *comm*. However, in Algol 60 all three expressions are evaluated anew at each iteration to (1) increment *j* and then (2) determine if iteration should continue.
- In some languages (Pascal, Ada, old Fortran), *step* can only be 1 or -1.
- In Algol 68 and Algol W, *j* is considered to be an integer variable declared locally throughout the body of the loop; it does not conflict with any declared variable of the same name. In other languages (e.g. Pascal), *j* must be declared in some enclosing scope as a normal variable of type **integer** or as a subrange. There may be further constraints on other uses of this *j*.
- If *j* is treated as a normally declared variable, then its value upon loop exit may be (a) considered undefined, or (b) equal to the value of *last*, or (c) equal to the last value obtained by successive incrementations (i.e., *first* + *k***step* if there have been *k* executions of the loop body).
- Prior to `f77`, versions of Fortran often did not specify the effect of an indexed loop when $F > L$. The reason was a wink at implementors: they could gain a slight efficiency in the generated machine code by placing the test at the end of the code for the loop body. Thus the body was always executed at least once, even when $F > L$.

(a) Add a **for** command to the abstract syntax and to the denotational semantics of IMP_0, the imperative language discussed in class (Watt, Example 3.6).

You will have to make decisions about the meaning of a **for** loop which will address the points above (and possibly other issues that arise). Note that the language has no **goto** or **break**, so issues of transfers *into* loop bodies or *out of* loop bodies do not arise.

(b) For each of the bulleted points above, describe what your `for` construct does, and explain how that semantic decision is reflected in the semantic rules.

4. Fixpoint Combinator

Define the combinator

$$\mathbf{Y} = (\lambda xy\,.\,y(xxy))(\lambda xy\,.\,y(xxy))$$

(a) Show that $\mathbf{Y}F$ is a fixed point for *F*, i.e., prove the *fixed point identity*:

$$\mathbf{Y}F = F(\mathbf{Y}F)$$

Show all reductions used and label them by type beta or eta (β or η). *HINT:* Write **Y** as **AA** and reduce **AA***F*. Also, remember how to parenthesize the expression *uvw*.

(b) Discuss the result of applying **Y** to the identity combinator $\mathbf{I} = \lambda x\,.\,x$.