

# MIDTERM EXAMINATION - CS130 - Spring 2003

---

Your **full** name:

Your UCSD ID number:

---

This exam is closed book and closed notes  
Total number of points in this exam: 120 + 10 extra credit  
This exam counts for 25% of your overall grade in the class

---

## Problem 1 (*20 points*)

Consider the following grammar:

```
<expression> -> <item>
               | <expression> + <expression>
               | <expression> * <expression>
```

```
<item> -> A | B | C | D | E
```

- a. (*5 points*) Show that this grammar is ambiguous.

There are two parse trees for the string  $A + B * C$ :

```
<expression> -> <expression> +           <expression>
               -> <expression> + <expression> * <expression>
               ->   <item>      +   <item>      *   <item>
               ->     A         +     B         *     C
```

and

```
<expression> ->           <expression>      * <expression>
               -> <expression> + <expression> * <expression>
               ->     A         +     B         *     C
```

- b. (15 points) Rewrite this grammar as an unambiguous grammar in which  $*$  has higher precedence than  $+$  and both are right-associative. Draw a parse tree for the string  $A + B + C * D * E$  to test your grammar.

One must separate the abstractions of  $+$  and  $\text{minus}$ . Also, the rules must be right recursive to implement right associativity.

```
<expression> -> <term> + <expr>
               | <term>
```

```
<term>         -> <factor> * <term>
               | <factor>
```

```
<factor>       -> <id>
```

```
<id>           -> A | B | C | D | E
```

**Problem 2** (20 points)

Compute the weakest precondition for the following statements and postconditions (assume all variables are integers):

a. (10 points)  $a = 2 * (b - 1) - 1 \quad \{ a > 0 \}$

$\{ b \geq 2 \}$

b. (10 points)  $\text{if } (a > 0) \text{ } a += 1; \text{ else } a -= 1; \quad \{ a > 0 \}$

$\{ a > 0 \}$

**Problem 3** (20 points)

a. (5 points) Consider the following fragment of C code:

```
int w=2;
int fun (int x) {
    int y = x + 2;
    return foo(y*2);
}
```

Make 10 statements about various binding and binding times regarding the elements of this code. Each valid statement you write is worth 0.5 point. For instance, five valid statements, which you cannot reuse, are:

"The meaning of the '{', '}', '(', ')', and ';' symbols are bound at language design time"

There are many possibilities, here are 10:

- The value of variable **y** is bound at run time
- The range of values for variable **y** is bound at language implementation time
- The type of variable **w** is bound at compile time
- The type of variable **y** is bound at compile time
- The return type of function **fun** is bound at compile time
- The meaning of the symbol **+** is bound at language design time
- The value returned by function **fun** is bound at run time
- The call to function **foo** is bound to a code address at link time
- The storage for variable **w** is bound at load time
- The internal representation of literal **w** is bound at language design time

- b. (10 points) Contrast static type binding and dynamic type binding. What are the advantages and drawbacks? List 2 languages that use static type binding and 2 languages that use dynamic type binding.

Static binding is done through implicit or explicit declarations, is **simple to implement**, and **allows static type checking** in most cases.

Dynamic type binding is **more difficult and costly to implement**, and **requires dynamic type checking**. However, it allows **greater flexibility** when writing programs and does not force the programmer to declare variables.

Java and C use static type binding, LISP and Javascript use dynamic type binding.

- c. (5 points) Give an example of a language construct in a statically typed language that requires dynamic type checking.

The C union

**Problem 4** *(20 points)*

- a. *(10 points)* What are the trade-offs between name type compatibility and structure type compatibility? Which type compatibility scheme do most language use ?

Name compatibility is **easy to implement but very restrictive**, while structure compatibility is **difficult to implement but more flexible**.

Most language use a combination of both (e.g. declaration compatibility).

- b. Multidimensional arrays can be stored in row major order, as in C, or in column major order, as in Fortran.

- (i) *(5 points)* Say why it is important for a programmer to know which way arrays are stored by the language

Knowing the way arrays are stored is important because it makes it possible to write more efficient code (e.g. by nesting loops in the correct order to increase locality and allow better cache reuse).

- (ii) (5 points) In a way similar to what we've done in class, write the access function for 3-dimensional column-major arrays. Assume an array `a`, with dimension sizes `m`, `n`, and `p`, with address of the first element `a[1,1,1]`, and with element size `element_size`.

`location(a[i,j,k]) = ...`

`location(a[i,j,k]) = address of a[1,1,1] + ((k-1)*m*n + (j-1)*m + i-1) * element_size`

**Problem 5** (20 points)

- a. Convert the following infix expressions into prefix expressions, assuming the standard operator precedences and associativities:

(i) (4 points)  $A * (B + C) - D$

$(- (* A (+ B C)) D)$

(ii) (4 points)  $A + ((B - C) * (D + E)) - F$

$(- (+ A (* (- B C) (+ D E))) F)$

(iii) (4 points)  $A - (((C) / D + (E - F)) / G)$

$(- A (/ (+ (/ C D) (- E F)) G))$

- b. Convert the following prefix expressions into infix expressions (use parentheses when needed): (4 points)

(i) (4 points)  $(- (+ (* A (/ B C)) D) E)$

$A * (B / C) + D - E$

(ii) (4 points)  $(* A (+ B (* D E)))$

$A * (B + D * E)$

**Problem 6** (20 points)

- a. (10 points) The Pascal **if** statement has the syntax:

*if Boolean\_expression then statement else statement*

while the Ada **if** statement is:

*if Boolean\_expression then statement else statement end if*

Comment on the relative merit of having an explicit terminator to the statement, such as the **end if** in Ada.

The **end if** in Ada removed the dangling-else problem, making it always explicit which **else** related to which **then**. Without and **end if** a statement, a simple grammar is ambiguous and some work is required to remove the ambiguity for easy parsing and compilation. Without the **end if** statement, the rule is that an **else** matches the closest unmatched **then**, which has poor readability and is error prone. In languages that do not have an **end if** statement, like C, a good programming practice is to use statement blocks (e.g. with { and } in C) which in effect emulated an **end if** statement.

- b. (10 points) What is the output of this C program? What is the value of **b** at the end of the program?

```
int f(int *b) {
    (*b)++;
    return *b;

int main() {
    int a=0,b=0;

    if (a && f(&b)) {
        a-=2;
    }
    if (a + f(&b) >= 1) {
        printf("--> foo\n");
    } else {
        printf("--> bar\n");
    }
    return 0;
}
```

The output is just "foo". The first call to function **f** never happens because its evaluation is short-circuited. At the end of the program the value of **b** is 1.

**Problem 7** EXTRA CREDIT (10 points)

Joe "I don't understand Prolog" Bob was asked to write a Prolog program to compute the length of a list, and this is what he turned in:

```
joeLength([],0).  
joeLength(_|Tail,Len) :-  
    joeLength(Tail,TailLen),  
    Len = TailLen + 1.
```

- a. (5 points) Why isn't Joe's code working?

The last line should be `Len is TailLen + 1` to be evaluated as an arithmetic expression rather than be a unification.

- b. (5 points) What would be the output of Joe's program for the following query (i.e. the value of L)?

```
?- joelength([1,2],L).
```

The output would be `0+1+1`.