CS 164
Spring 1997

Second Midterm Exam

1 (25 points)
Consider the following attribute grammar:

Start	->	*Prog*	{ *Prog.In = NewEnv();* *Prog.Whole = Prog.Out*
Prog	->		{ *Prog.Out = Prog.In}*
	|	$Prog_1$ *Class*	{ $Prog_1$*.In = Prog.In*; *Class.In =* $Prog_1$*.Out*; *Prog.Out = Class.Out*; $Prog_1$*.Whole = Prog.Whole*; *Class.Whole = Prog.Whole*;}
Class	->	**class ID is** *Features* **end**;	{ *Assert*(!*Defined*(*Class.In, ID*)); *Class.Out = Add*(*Class.In, ID*); *Features.Whole = Class.Whole*;}
Features	->		
	|	$Features_1$ *OneFeature*	{ $Features_1$*.Whole = Features.Whole*; *OneFeature.Whole = Features.Whole*;}
OneFeature	->	$\mathbf{ID_1 : ID_2}$	{ *Assert*(*Definded*(*OneFeature.Whole,* ID_2));}

This grammar describes a subset of the COOL language in which a class is a collection of features of some type. As in COOL, forward declaration of classes are allowed and redefinitions of classes (same name used more than once) are disallowed.

The environment functions have the following meaning:

- *NewEnv()* - creates a new, empty environment
- *Add(env, sym)* - creates a new environment by adding symbol *sym* to environment *env*.
- *Defined(env, sym)* - true if symbol *sym* is defined in environment *env* and false otherwise.
- *Assert(cond)* - checks that condition *cond* is satisfied.

a) *(6 points)* Fill in the following table

Attribute	Inherited or Synthesized	Brief Description of its Purpose
In		
Out		
Whole		

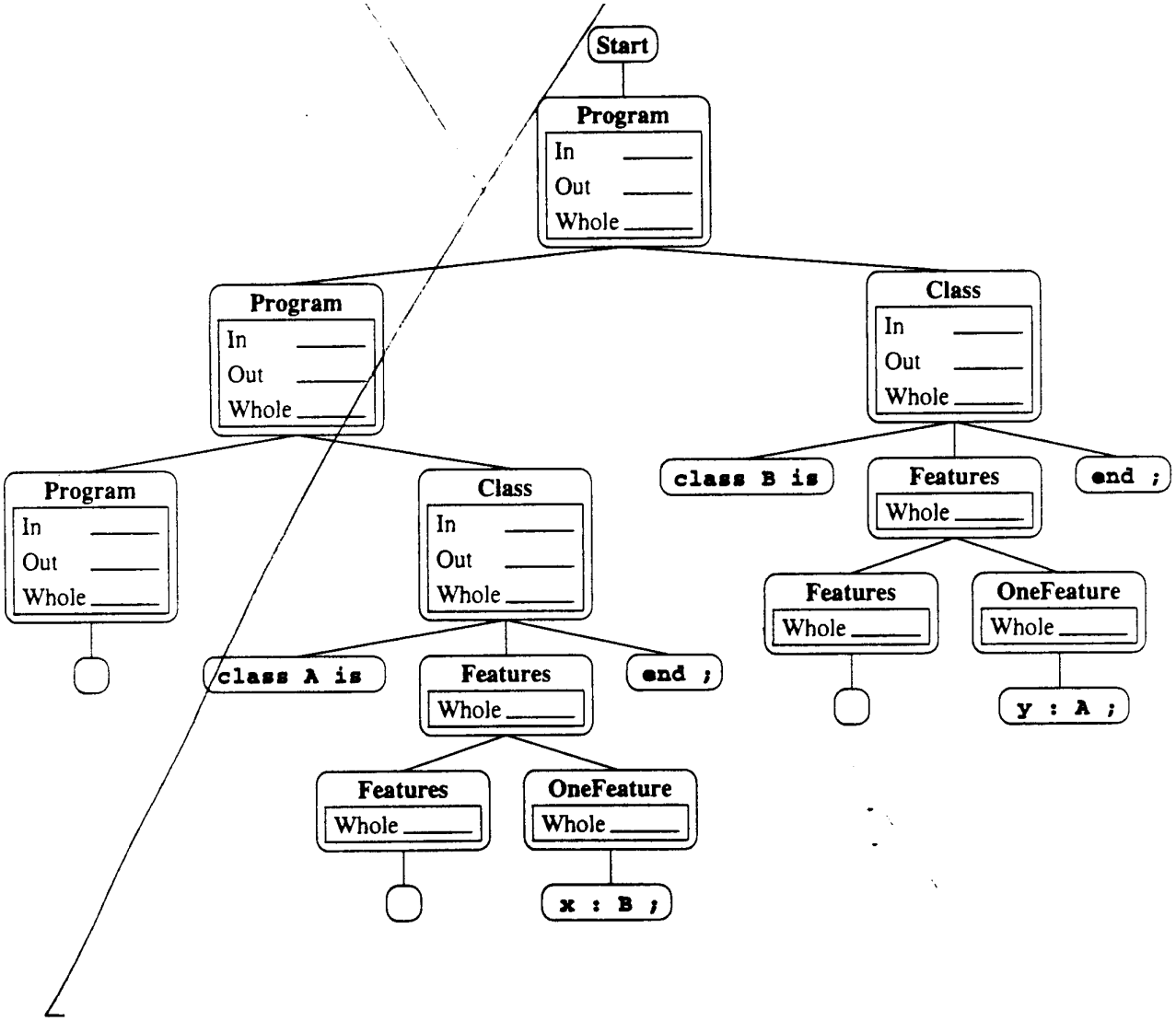
b) *(1 point)* Is this grammar L-attributed? Why or why not?

Yes No

c) *(1 point)* Can this grammar be evaluated with a single pass over the abstract syntax tree? Why or why not?

Yes No

d) *(8 points)* Compute attribute values for all nodes in this sample abstract syntax tree:



[The question continues on the next page.]

e) *(7 points)* Simplify the attribute rules so that no forward declarations are allowed. In other words, at each point in the source file you only allow class features to have types that have been defined prior to that point. Use the grammar below as a template: **fill in all the blanks and cross out all unecessary attribute rules.**

Start	->	*Prog*	{ *Prog.In = NewEnv();* *Prog.Whole = Prog.Out}*
Prog	->		{ *Prog.Out = Prog.In;}*
	|	$Prog_1$ *Class*	{ $Prog_1$*.In = Prog.In;* *Class.In =* $Prog_1$*.Out;* *Prog.Out = Class.Out;* $Prog_1$*.Whole = Prog.Whole;* *Class.Whole = Prog.Whole;}*
Class	->	**class ID is** *Features* **end**;	{ *Assert*(!*Defined*(*Class.In, ID*)); *Class.Out = Add*(*Class.In, ID*); *Features.Whole = Class.Whole;*

			_____}_____
Features	->		
	|	*Features*$_1$ *OneFeature*	{ *Features*$_1$*.Whole = Features.Whole*; *OneFeature.Whole = Features.Whole*; _____ _____}
OneFeature	->	**ID$_1$: ID$_2$**	{ *Assert*(*Definded*(*OneFeature.Whole, ID*$_2$)); _____)

f) *(2 points)* Gve one advantage and one disadvantage of allowing forward declarations in a programming language. (The disadvantage should not be a negation of the advantage!)

Advantage:

Disadvantage:

2 (10 points)

Indicate whether each of the first three statements is true or false by circling either T or F. For each true/false statement, give a brief explanation (once sentence or an example) justifying your answer. Also answer the last four questions.

T **F** A grammat that is left-recursive or is left-factored cannot be an LL grammar.

T **F** A compiler does not need a symbol table to implement dynamic scoping.

T **F** If a programming language does not allow recursive routines, all storage can be allocated statically.

What is a control link (i.e. to what does it point)?

What is its purpose in a runtime system?

What is an access link (i.e. to what does it point)?

What is its purpose in a runtime system?

3 (20 points)

Use the pseudocode below to answer the following questions:

```
class A {
   method() : Char {return 'A';}
}

class B inherits A{
   method() : Char {return 'B';}
}

class C inherits B{
   method() : Char {return 'C';}
}
```

```
class D inherits B{
      method() : Char {return 'D';}
}
```

a) What value or values could **x.method()** return, if **x** is declared to be of type **B?**

b) Which of the following method definitions would be rejected by the type checker?

1. method1(x : B) : A {return x;}

2. method2(x : A) : B {return x;}

3. method3(x : B) : A {x <- method3(x); return (new A);}

4. method4(x : A) : B {x<- new B; return x;}

Consider an expression in a strongly-typed language where an aggregate expression's value may be either of two component values. For example,
   if <condition> then <expr1> else <expr2>;
Let T1 be the type of expr1 and T2 be the type of expr2.

There are at least three different ways that the type of an aggregate expression could be defined. (Any rule for the aggregate type which admits all values of the component types will result in a sound type system.)

- Define T1 u T2 to be the least upper bound of T1 and T2. The least upper bound is a type T3 that is an ancestor of bothe T1 and T2, and for which no type T4<=T3 exists that is also an ancestor of both T1 and T2. COOL uses T1 u T2 as the type of an aggregate expression of T1 and T2. T = T1 u T2 works because T1 and T2 must inherit from T.
- Use a universal parent class like Object. That solutiion works because all types (including T1 and T2) inherit from Object.
- Construct a new type Tunion which is the union of types of T1 and T2 (T1 u T2) (i.e. a type conforms to Tunion if it conforms to either T1 or T2).

c) Referring to the pseudocode at the beginning of the question, what is A u B?

d) given this piece of code: if C1 then (new C) else (new D);

what is the resulting type under each of the three possible type rules listed above (using the code sample at the beginning of this section)?

T1 u T2 _____ Object _____ T1 u T2 _____

e) Give one advantage and one disadvantage of using the rule T = T1 u T2 instead of T1 u T2.

Advantage:

Disadvantage:

f) Give on advantage and one disadvantage of using the rule T = Object instead of T1 u T2.

Advantage:

Disadvantage:

Again assume we have a strongly-typed language. Let x be an attribute declared as type B (again using the inheritance hirarchy defined by the pseudocode at the beginning of thie question). After executing the following statement:

```
      x <- new C;
```

g) What is the static type of x, assuming COOL's typechecking rules?

h) What is the dynamic type of x, assuming COOL's typechecking rules?

i) Can the static type of a variable be a subtype of (inherit from) the dynamic type of that variable?

j) Can the dynamic type of a variable be a subtype of (inherit from) the static type of that variable?

4) (25 points)

We want to implement a symbol table using a hash table lookup with linked lists stored at the buckets. The symbol table should support the functions add-id, lookup-id, enterscope, and exitschope. The partiall C++ declarations for the data structures follow:

```
#define N 5 // N = size of hash table

typedef char * Symbol; // Symbol = char *

struct ListEntry {
 ListEntry *next; // next entry in linked list
 /* possibly other data or data structures */

 Symbol id; // name of id.
Attributes *attrs; // type info and other attributes. Assume
 // Attributes is another struct defined somewhere.
};

class SymTab {
  private:
 ListEntry *hashTable[N]; // An array of ListEntry pointers
 /* possibly extra data or data structures */

  public:
 void add-id(Symbol id);
 ListEntry *lookup-id(Symbol id);
 void enterscope();
 void exitscope();
}
```

a) Give one advantage and one disadvantage that a hash-based symbol table has over a list-based symbol table.

Advantage:

Disadvantage:

Parts (b) - (f) ask you to describe the implementation of the symbol table. Describe your implementation in words, not in code or pseudocode. As an example, a description for add-id (that is not correct) is "First hash the Symbol to find the hash bucket. Make a new ListEntry for the symbol and add the new ListEntry to the end of the list at the hash bucket."

Assume you have an appropriate hash function **int hash(Symbol id)**.

b) Are any new data members or data structures needed to implement the functions? If so, what are they?

c) Describe the implementation of the function **void add-id(Symbol id)**.

d) Describe the implementation of the function **ListEntry *lookup-id(Symbol id).**

e) Describe an efficient implementation of the function **exitscope()** (i.e. one that does not always scan the entire hash table).

f) Describe the implementation of the function **enterscope()**.

g) Draw a picture of your SymTab used for lexical (static) scoping for the following Cool code fragment:

```
 foo(a:int, b:real):object is
 let b:int <- 2, c:string in "new scope"; end;
```

```
let b:int <- 3 in "new scope"; end;
(* draw SymTab for this point in the program *)
.
.
.
end;
```

Assume the hash table size (N) = 5, that the hash function only considers the first letter of the identifier and maps letters as follows: a -> 0 mod N, b -> 1 mod N, ..., z -> 25 mod N

When drawing ListEntry's, you may ignore the attrs field. Here is an example (assume the id name is foo here):

```
+-------------------------------------------------------------------+
| foo | your extra data, if any | o--|---> next
+-------------------------------------------------------------------+
```

In addition, to denote any deleted ListEntry's or other data structures, please just cross them out (do not erase them).

BODY>