CSCI 565: Compiler Design and Implementation
Spring 2006

Midterm Exam Solution
Feb. 28, 2006

Problem 1: Attributive Grammar and Syntax Directed Translation [30 points]

A language such as C allows for user defined data types and structures with alternative variants using the union construct as depicted in the example below. On the right-hand-side you have a context-free-grammar for the allowed declarations which has as statring symbol a struct_decl.

```
typedef struct {
  int a;
  union {
    int b;
    char c;
    double d;
  } uval;
  int e;
} A;
```

base_type	->	*int* | *double* | *char*
base_type_decl	->	base_type *identifier*
field_decl	->	base_type_decl
	|	union_decl
	|	struct_decl
field_list	->	field_list field_decl
	|	field_decl
	|	ε
struct_decl	->	*typedef struct* { field_list } *identifier*
union_decl	->	*union* { field_list } *identifier*

Answer: This question assumes you understand what a union is (variant recond in Pascal) and how the compiler organizes the layout for a union and struct in memory. Essentially all fields of a union share the same storage since only one can be active at a time. For the example given all the fileds b, c and d use the same storage space. As such the compiler allocates to the union the size corresponding to the largest field of the union, thereby ensuring there is enough space for all of them. This suggests that your set of attributes must keep track if you are calculating the starting addresses of each field in a struct or in a union. In a struct you simply add up all the sizes of the fields whereas in a union you determine the maximum over the sizes of the fields of the union.

(a) [10 points] When accessing a field of a union the compiler must understand the offset at which each element of the union can be stored relative to the address of the struct/union. Using the context-free grammar depicted above derive an attributive grammar to determine the offset value for each field of a union. Be explicit about the meaning of your attributes and their type. Assume a double data types requires 8 bytes, an integer 4 and a character 1 byte. Notice that unions can be nested and you might want to rewrite the grammar to facilitate the evaluation of the attributes.

Answer The basic idea of a set of atributes is to have a inhereted attributes to track the current start of each field of either a union of a struct and another synthesized attributes to determine what the last address of each field is. The synthesized attribute named start, starts from 0 and is propagated downwards towards the leaves of the parse tree and indicates at each level the current starting address of a given field. The inhereted attribute, called end, determines where each field ends with respect to the base address of the struct or union. Because in the struct we need to add up the sizes of the fields and in the union we need to compute the maximum size of each field we have another inhereted attribuest, called mode, with the two possible symbolic values of AND and OR.

In terms of the attribute rules we have some simple rules to copy the inherited attributes downwards and the synthesized attributes upwards as follows:

```
field_decl -> union_decl    { union_decl.start = field_decl.start; field_decl.end = union_decl.end; }
field_decl -> struct_decl   { struct_decl.start = field_decl.start; field_decl.end = struct_decl.end; }
field_decl -> base_decl     { base_decl.start = field_decl.start; field_decl.end = base_decl.end; }
```

At the bottom of the parse tree we need to make the assignment of the end end attributes given an imput start attributes as follows:

```
base_decl -> int identifier          { base_decl.end = base_decl.start + 4; }
base_decl -> char identifier         { base_decl.end = base_decl.start + 1; }
base_decl -> double identifier       { base_decl.end = base_decl.start + 8; }
```

Finally we have the rules that do the propagation of the attributes in the manner that takes into account the fact that we are inside a union or a struct. The first set of rules is a simple copy of the values between the field_decl_list and a single field_decl:

```
field_decl_list -> field_decl        { field_decl.start = field_decl_list.start; field_decl_list.end = field_decl.end; }
```

The real work occurs during the recursive rule:

```
field_decl_list_0 -> field_decl_list_1 field_decl {
  field_decl_list_1.mode = field_decl_0.mode;
  field_decl_list_1.start = field_decl_list_0.start;
  if (field_decl_list_0.mode == AND) then
    field_decl.start = field_decl_list_1.end;
    field_decl_list_0.end = field_decl.end;
  else
    field_decl_list_1.start = field_decl_list_0.start;
    field_decl_list_0.end = MAX(field_decl_list_1.end,field_decl_.end);
  end if
}
```

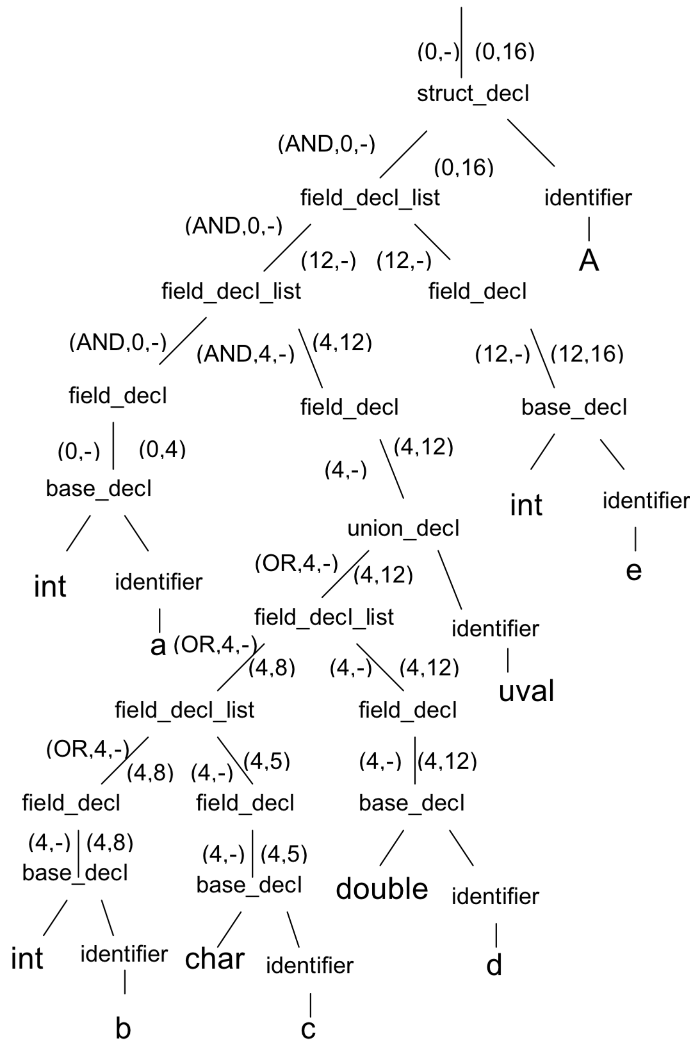
At last we need the rules that set at a given stage the mode attributes:

```
struct_decl -> struct { field_decl_list } identifier {
        field_decl_list.mode = AND;
        field_decl_list.start = struct_decl.start;
        struct_decl.start = field_decl_list.end;
}

union_decl -> union { field_decl_list } identifier {
        field_decl_list.mode = OR;
        field_decl_list.start = union_decl.start;
        union_decl.start = field_decl_list.end;
}
```

(b) [10 points] Show the result of your answer in (a) for the example in the figure by indicating the relative offset of each filed with respect to the address that represents the entire struct.

Answer: In the figure below we illustrate the flow of the attributes downwards (to the left of each edge) and upwards (to the right of each edge) for the parse tree corresponding to the example in the text. In the cases where the mode attribute is not defined we omit it and represent the start and end attributes only.



(c) [10 points] In some cases the target architecture requires that all the elements of a structure be word-aligned, i.e., every field of the struct needs to start at an address that is a multiple of 4 bytes. Outline the modifications to your answer in (a) and (b).

Answer:

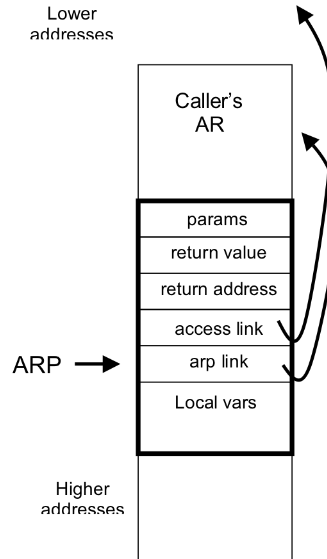
In this case the rules for the base declaration need to take into account the alignment of the inherited start attribute and can be recast as follows using the auxiliary function align_address that returns the next word-aligned address corresponding to its input argument.

```
base_decl -> int identifier        { base_decl.end = align_address(base_decl.start) + 4; }
base_decl -> char identifier       { base_decl.end = align_address(base_decl.start) + 1; }
base_decl -> double identifier     { base_decl.end = align_address(base_decl.start) + 8; }
```

Problem 2: Activation Records and Run-Time Environments [25 points]

We used the abstraction of the activation record to save run-time information about where to find the non-local variables (via the access-link) and also the return addresses of procedures and functions.

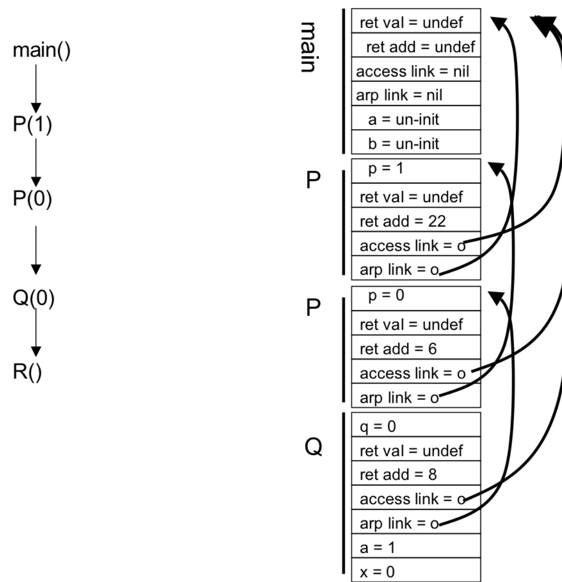
```
01:  procedure main (){
02:    int a, b;
03:    procedure P(int p)
04:    begin
05:      if (p > 0) then
06:        call P(p-1);
07:      else
08:        call Q(p);
09:    end
10:    procedure Q(int q)
11:      int a,x;
12:      procedure R()
13;      begin (* R *)
14:        print(a,x);
15:      end (* R *)
16:    begin (* Q *)
17:       a = 1; x = 0;
18:       if (q == 1) return;
19:       call R();
20:     end (* Q *)
21: begin (* main *)
22:   call P(1);
23: end   (* main *)
```



(a) [5 points] Draw the call tree for the code enclosed starting with the main procedure and ignoring library functions.

(b) [10 points] Draw the configuration of the stack in terms of ARs indicating the values for the fields corresponding to parameters and local variables as well as the access link and arp link (the stack pointers) when the execution reaches the statement on line 18. Use the organization for your activation record as shown above using the return address value value as ***the same line*** as the call statement in the source code (Obviously after the call you do not execute the same call again, so the return is to the end of the same line in the source code). Justify the values for the access link field in each AR.

Answer: See the figure below for (a) and (b).



(c) [5 points] Do you think using the display mechanism in this particular case would lead to faster access to non-local variable? Please explain.

Answer: For the accesses to both P and Q we only need a single indirection via the access link so using the display does not lead to any inprovement in terms of access time. For the procedure R, however, we would need 2 links to access the variables int a, and int b local to the main procedure. Only for that procedure R would the display make any difference. In practice and given that in reality R makes no accesses at all to non-local variable we actuall do not need the display.

(d) [5 points] For procedures at the leaves of the call tree, it is possible to allocate them statically rather then on a stack. Is this true, and if so under what circumstances? Suggest a simple compiler algorithm for allocating the AR of a given procedure on the stack or statically.

Answer: Yes, it is possible to allocate the corresponding AR statically, *i.e.* on a specific location in the address space. The reasons being that since there is no recursion between these procedures and any other procedures in the code there is a sone-to-one mapping of dynamic instances of the local variables and storage.

A quick and simple algorithm is to compute the call-graph first. Then identify the cycle-free regions of the call-graph that are the closest to the “leaves” of the call graph. If there is one cycle in the call-graph – meaning there is possibly recursion, all the nodes in the call graph above (in the direction of the root) including the nodes in the cycles are allocated on the stack and all the node in the lower portionof the call graph that is cycle-free can be allocated statically.

Problem 3: Compilation Techniques for Object-Oriented Languages [15 points]

In object-oriented languages the main issue of compilation is message dispatch or method invocation keeping the offset of the object data fields consistent with the various views of the methods in the presence of inheritance.

(a) [5 points] Describe the basic implementation technique to locate at run time the data structure that contains the references to a method's code in order to execute a given method.

Answer: This is the object block or simply the object's data object for the lack of a better name. It consists of a first filed with a pointer to a table of method pointers followed by the various data fileds corresponding to the data members of each instance of the bojects of that class. At run-time the sequenc eof instructions first fetchs the table pointer (at offset zero of the object's block) and then adds the offset corresponding to the specific method to be invoked. At this stage the generated code has the address of the method to be invoked and can now begin the pre-call sequence.

(b) [5 points] Single inheritance can be implemented used prefixing of method tables. In order to build these prefix tables the compiler will have at compile time to examine the program to understand what inherits from what. Suggest a simple compiler algorithm to determine the layout of the method tables.

Answer: A simple compiler algorithm will first build the inherits-from graph. Under the assumption that this graph is acyclic (a cycles means a class inherits from itself which is disallowed and you should not worry about this pathologic case), you just traverse the class inheritance graph from each class to its ancestors and accumulate along the way the number of methods defined in each class and not a redefinition of other methods. In the end you'll have computed for each class the number of methods they can directly invoke and that is the size of the method table. Notice that this works well in the case of single inheritance, multiple inheritance, where the classs inheritance graph is a DAG is slightly more complicated as you need to take care of repetitions (multiple paths) along the inheritance graph.

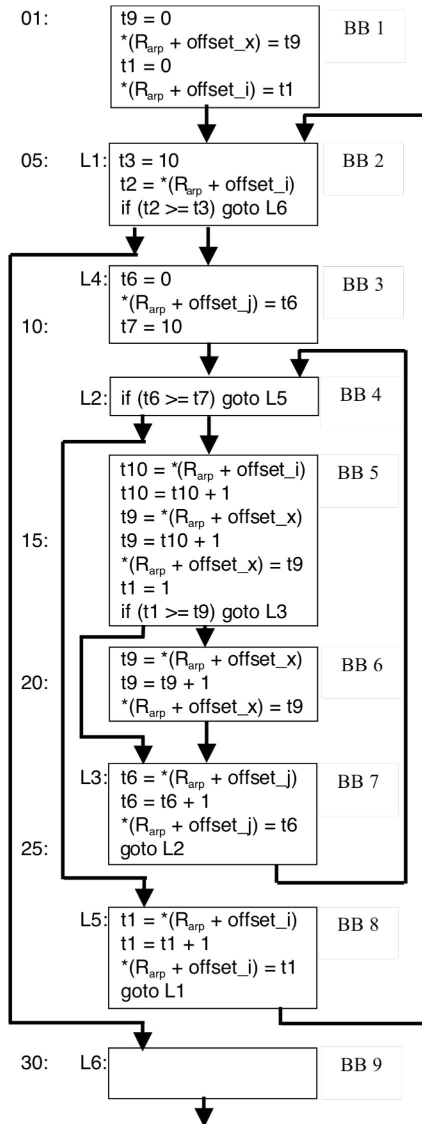
(c) [5 points] In the case of single inheritance how does the compiler determine the offset of each field in each object so that it can compile the code of each method correctly?

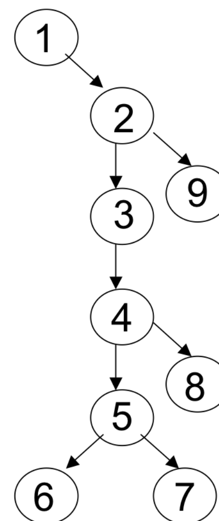
Answer: Again this is the simplest case and using the class inheritance graph the compiler builds the layout of each block by accumulating the data fields (and the corresponding offsets) for the current class with the classes it inherits from in a linear fashion following a path from each class to its ancestors. When compiling the code it simply looks up the offset of each data field and uses the current value of self (at run-time) to gain access to the objects data field values.

Problem 4: Control-Flow Graphs and Basic Optimizations [35 points]

(a) [10 points] Determine the Control-Flow-Graph (CFG) for this code identifying each basic block by the line numbers corresponding to the instructions in them.

Answer: Below is the original code with the basic blocks highlighted. In this construction we have used the labelling algorithm for the identification of the leaders of each basic block. Notice that in some cases there are edges between basic blocks that correspond not to explicit jumps or target to jump but by the implicit transfer of control between one instruction and another instruction that is the target of a jump. On the right is the dominator tree for this CFG.





(b) [10 points] Identify the dominator tree for the CFG drawing the tree explicitly.

Answer:
The dominator tree (above on the right) is constructed by recognizing which nodes dominate which nodes. The definition of dominance states that for a node p to dominate a node q then every path from the entry (in this case the node 1) to q must necessarily pass through p. For example node 5 dominates 6 and also 7 since every path from the entry to 6 or 7 must pass through 5.

(a) [10 points] Identify the back edges and the corresponding loops in the code.

Answer: In order to find the loops in the code we need to find the "back edges", i.e. the edges in the CFG such that the node pointed o by the head dominates the node at the tail of the edge. For our example, the back edges are (8,2) and (7,4) to which correspond the loops (4,5,6,7) and (2,3,4,5,6,7,8). As it can be seen the set of nodes corresponding to the first back edge is a subset of the set of nodes corresponding to the second back edge, hence the loop corresponding to the first back edge is nested within the loop corresponding to the second back edge.

(b) [5 points] Is it possible to move the sequence of instructions in lines 12 and 13 out of the innermost loop? Why, why not?

Answer: Yes, since this computation uses the variable "i" which is loop invariant with respect to the innermost loop, a loop on the "j" variable, we can move these instruction to the basic block 3.