Assignment 4
Structs and Arrays

15-411: Compiler Design
Ruy Ley-Wild (`rleywild@cs`)

Due: Thursday, October 20, 2008 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Thursday, October 20. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1

[20 points]

Consider the beloved C struct

```
struct C {
  char cFoo;
  int iBar;
  double dNaN;
  struct D *next;
  short sBaz;
  char cQux;
  long long llRockets;
};
```

1. Conforming to the x86-64 ABI, describe the layout of the structure by giving the offset of each field, the total size of the structure, and its alignment requirement.
2. If we violate the ABI by reordering fields, we can give a more space-efficient layout of the structure. Describe a minimum-space layout that still obeys the x86-64 alignment restrictions.
3. Unlike C, languages such as Java and C# don't impose a strict ordering requirement on structure members; the runtime is free to reorder the fields as it sees fit. Give a simple algorithm to compute an ordering of fields which gives the smallest structure size.
4. Explain whether your algorithm from part (c) work if structures had bitfields (anywhere between 1 and 32 bits).

Problem 2

[20 points]

1. Unlike *L4*, C allows fixed-size arrays and unions in structs. Give an SML structure implementing the following signature, `STRUCT`. `sizeof` should take a field type and return its size in bytes. `field_byte_offset` should take a struct (represented as a list of named fields) and a field identifier and compute the offset of a field in the struct; assume that any given struct has been checked for duplicate field names. Int's are 4 bytes; the size of a struct $\{f_1{:}\tau_1; \ldots f_n{:}\tau_n;\}$ is computed by traversing the fields from left to right, adding padding as necessary so that alignment restrictions on the fields are satisfied. Int's are aligned at 0 mod 4; an array of type $\tau[n]$ is aligned as τ is aligned; a union is aligned to ensure all of its fields are aligned; structs are aligned according to their most strictly aligned field. Padding may need to be added at the end of a struct so that its total size is a multiple of its most strictly aligned field.

```
signature STRUCT =
sig
  datatype field_type = Int
                      | Array of field_type * int
                      | Union of fields
                      | Struct of fields
  withtype fields = (string * field_type) list

  val sizeof : field_type -> int
  val field_byte_offset : fields -> string -> int
end
```

2. Suppose we wanted to extend the definition of *L4* to permit fixed size arrays and unions so they can be directly embedded in structs. Describe in detail how to modify the definition as given in the Lab 4 handout to accomodate this extension. This should include, as you find necessary, extended or modified syntax, static semantics including typing rules, and rules for program execution. You may assume a C-like unsafe model of execution where the results of certain operations, such as accessing an array out of bounds, are undefined.

Problem 3

[20 points]

1. Although costs for memory are dropping every year, memory consumption is still a problem, especially in large applications. Choosing the right types for variables can play an important role in performance in terms of both memory and speed. C provides many integral data types (`char`, `short`, `int`, `long`, `long long`) and floating point types (`float`, `double`, `long double`) but doesn't give good specifics on their sizes. In contrast, both C# and Java provide the same types[1] including absolute sizes so it is easier to pick a type that works in multiple environments. GCC and MSVC have extensions to allow integer structure fields to be sized explicitly in bits (up to 64). Since architectures generally do not provide instructions for writing arbitrary numbers of bits to arbitrary locations, these compilers implement writes to and reads from bitfields using shifting and masking.

   Consider the following C code:

   ```
   extern struct {
     int x:17;
     int y:15;
   } s;

   int setfield(int a) {
     s.y = a;
     return s.x;
   }
   ```

   Implement `setfield` in x86-64 assembly using as few instructions as possible. How would you implement basic arithmetic operations in general?

2. In C, the `volatile` keyword can be appended to the type of a declaration to indicate that the value being declared may change at any point in the program's execution, for instance, through modification by another thread or by the hardware. The compiler must carefully avoid applying optimizations that assume that a `volatile` variable has a fixed value.

   Give a piece of code and an optimization that cannot be applied to the code because a variable involved in the computation is `volatile`.

3. Considering the way in which writes and reads involving bitfields are implemented by the compiler, describe a race condition that may arise in a multi-threaded program in which multiple threads share access to a struct with `volatile` bitfields, even when each bitfield has a lock which is always used to synchronize read and write access to it. As a compiler-writer, how would you address this problem?

[1]Except for unsigned numbers in Java