MC97 Lab 6: The compiler to end all compilers

Due May 15, 1996; Max Hailperin

Address any outstanding grading comments from prior labs and do anything you wish you had done on prior labs, but didn't have time to do. Do as many of the following as seem worthwhile; I strongly urge you to at least do some form of one-dimensional arrays and the comparison with cc. Note that these are not in order; you may do items later on the list without doing earlier ones.

- Add arrays. Possible variations include
  - One-dimensional *vs.* multidimensional
  - Local *vs.* global
  - Restrictions on size specification: constant *vs.* any expression which evaluates to a constant *vs.* any arbitrary expression
  - Passing of arrays by reference *vs.* by copying *vs.* not at all.

  There is a section below discussing some of the issues.

- Compare the quality of your object code with that of the `cc` C compiler (both with and without `cc`'s `-O` (optimize) flag). At a minimum, compare the two assembly language files by hand for interesting differences; to get `cc` to produce assembly language rather than a binary object file, use the `-S` flag. You may also want to do a quantitative performance comparison; doing so requires overcoming a number of grungy little technical details, so see me if you are interested in working on this.

- Add the boolean operators `&&`, `||`, and `!`. I'd advise keeping your code generation for these straightforward, even if far from optimal, unless you are particularly bold: trying to do it right can be quite a tar pit.

- Add the `float` type, with appropriate type checking and coercion and overloaded operators.

- If you didn't do this in lab 5, arrange that the evaluation of the *last* argument to a procedure is targeted directly to the argument register, rather than to a temporary register.

- Add strings. (What should that mean?)

- Add any other language features (e.g. borrowed from C) or optimizations you want.

- Improve error handling (reporting and/or recovery).

- Add register spilling. (This will require you to allocate a bigger stack frame, much as for local arrays. See the section below on implementing arrays for details of this.)

- Add handling for more than four words of arguments (see the on-line documentation for the calling convention).
- Provide documentation of the language you are compiling, of implementation strategies and conventions, and of implementation limitations.

Arrays

Global arrays can be created much as other global variables, they are simply larger (some multiple of four bytes, rather than always four bytes). Instead of using lots of "`.word 0`" lines in a row, it would be better to use "`.space` *n*", where *n* is the number of *bytes* of space you want (four times the number of array elements).

Access to the array elements would be with `lw` and `sw`; the only difference is that you have to add the array index times the size of the elements to the base address. For example, if we wanted to do `a[i] = a[j];` where `i` and `j` are in registers `$s0` and `$s1`, we might emit the following code:

```
la $t0, a           ; load a's address into $t0
mul $t1, $s1, 4     ; set $t1 to 4 * j
addu $t0, $t0, $t1  ; add (unsigned) to get address of a[j]
lw $t0, ($t0)       ; load contents of a[j]
la $t1, a           ; again, a's address
mul $t2, $s0, 4     ; four times i
addu $t1, $t1, $t2  ; address of a[i]
sw $t0, ($t1)       ; store
```

For global arrays it is conventional to insist on constant sizes. You can easily loosen this to allow any expression that evaluates to a constant, assuming you are doing constant folding anyway. For local arrays this restriction of constant array sizes is also sufficiently simplifying that in this handout I'll only explain that case. On the other hand, allowing local arrays to have dimensions computed at runtime is quite handy; if you are interested in this, ask me and I'll explain how to handle it.

If (as we're assuming) local arrays have constant sizes, then we can translate

```
f(){
  int a[100], b[20];
  ...
}
```

by increasing the stack frame size by 480 bytes (i.e., increase the amount we subtract from `$sp` in the procedure prolog and add to `$sp` in the procedure epilog by 480 bytes, so it is 560 instead of 80) and then translating the procedure as if it were

```
f(){
  int *a, *b;
```

```
  a = $sp + 80;   // by $sp I mean that (stack pointer) register
  b = $sp + 480;
  ...
}
```

Note that the pointers `a` and `b` can be stored in registers just like any other local variable; only the actual arrays themselves need to be allocated on the stack.