Project 3 – Local Register Allocation

Due Date: June 16, 2009 at midnight

Description: In this third part of the compiler project, you will be asked to implement a simple local register allocator and rewrite a sequence of three-address instructions using a set of registers rather than symbolic variables. In this project you assume the location of the local variables, all assumed to be local variables to the procedure you are generating code for, are indeed local variables found at specific offset of the Activation Record Pointer (ARP). As such whenever you need to load a specific variable value from memory you need to use the correct offset, compute the address and then load the value of the variable. The details on how to generate code are described below.

The project shall be developed individually. You are encouraged to discuss with other students, ask teachers, search for ideas on the Internet. But, do not copy code. The solutions will be tested in a UNIX environment as with previous projects using a set of known test inputs. The evaluation of the solutions is based on these and other set of tests, which are not known to you in advance. Your project can be developed in many environments. But, when the solution is delivered, it must be possible to generate an executable code using a UNIX `makefile`. See the instruction on how to turn-in your projects also described below.

Register Allocation for Three-Address Instructions. In this project, you are given a sequence of intermediate representation instructions using the three-address instruction format described in class and translate that code into the same intermediate representation but using a limited set of physical registers. Given a set of `k` physical register, say register r_0 through r_k, you need to translate an instruction that uses temporary variables and local variables into the same instruction using registers. All identifier symbols in instructions that begin with the "t" character are considered temporary variables. All other identifier symbols are considered as local variables for which there is an auxiliary description associating an integer value, its offset relative to the ARP, where the variable is located in the Activation Record (AR). You may consider that the input sequence of instructions does not contain any identifier symbol that begins with the "r" character as these are reserved for the physical registers.

Register Allocation Algorithm: In this project you are asked to use the local register allocation algorithm described in class for a sequence of instructions inside a given basic block. You will be given in the command the number of registers as a numeric integer value indicating the maximum number of registers used. You will assume that all registers are identical and all values the symbolic and local variables manipulate are of the same type, say integer. There is also a file specifying the integer offset of the ARP for each of the local variables. As such an access (in read mode) to the variable "a" can be translated in the following sequence of three-address instructions assuming the offset of "a" is 8 with respect to the base of the ARP:

```
r1 = ARP + 8        // here r1 is the address of "a" in the AR
r1 = *r1            // r1 now has the value of "a"
```

You may assume all local variables the program manipulate have a unique specification of the offset and that all offsets are positive. The example in the figure below illustrates the mapping of the input

sequence of instructions on the left to the sequence of instructions on the right assuming the mapping of local variables to offsets as follows: `{a,b,c} = {4,8,12}`.

```
t1 = a + b              r0 = ARP + 4        // r0 has address of a in AR
                        r0 = *r0            // r0 has value of a
                        r1 = ARP + 8        // r1 has address of b in AR
                        r1 = *r1            // r1 has value of b
                        r2 = r0 + r1        // r2 has value of t1
t2 = t1 + 1             r1 = r2 + 1         // reuse r1; r1 now has t2
                        r2 = ARP + 12       // r2 has address of c
                        r2 = *r2            // r2 has value of c
t3 = t2 + c             r0 = r1 + r2        // reuse r0; r0 now has t3
a  = t3                 r2 = r0             // r2 has the value of a
                        r1 = ARP + 4        // r1 has address of a in AR
                        *r1 = r2            // storing a as new value of a in AR
                                            // as a has been modified in this block
```

As part of the register allocation you need to keep track of which local variables has their value written and when the corresponding register needs to be reused by the algorithm for another local variable you need to generate a "store" operation where you update the value of the variable in the Activation Record (AR). This operation might require an additional register. The figure below illustrates this case assuming that variable "a" assigned to register `r0` which needs to be used for another variable has been modified (written) prior to this instruction.

```
r0 = …          // variable "a" was assigned to r0; needs to be reused
r1 = ARP + 8    // compute the address of "a"
*r1 = r0        // saving the value of "a" in memory
r0 = …          // here the register r0 is being assigned to another variable
```

Output of your project

As the output of your project you will generate to the output the number of loads and store operations as well as the sequence of three-address instructions using registers. The output is redirected to the output stream and the indication of the number of loads and stores uses the syntax below illustrated for the case where we have 3 loads and 1 store. Note that whatever variables have been modified at the end of the sequence of instructions, they need to be stored in the corresponding activation record.

```
Number of Loads: 3
Number of Stores: 1
```

Should you not be able to generate code as you do not have a minimal number of register, say 1, you generate an empty sequence of instructions at the output proceeded by an indication that you have done zero load and store operations.

Input to your project

As the input to your project you will receive the sequence of three-address instructions in a specific file, say test1.in as the first argument to your executable. The second argument will indicate the number of registers and the third and last argument will indicate the file where the offsets of the local variables are specific. There is no specific order to the offsets of the variables. We will provide a C source code template to parsing the values of the input correctly so that you need not worry about the parsing of the input instructions or the values of the offsets. The example below illustrates

the intended invocation of your project for an input file name "test1.in" with 3 registers and using the mapping of local variables to offset in the file "map1.in"

```
./proj3 test1.in 3 map1.in
```

The mapping file has a simple syntax as illustrated below

```
a     4
b     8
c     12
```

Turn-in Instructions: As with the second programming assignment you will turn in your project on Fénix. Please make sure you have a group and can submit your files. If you are student number `XYZ`, you need to create a file named `XYZ.proj3.tar.zip` created using the `zip` and `tar` utilities, and make sure that the commands

```
unzip XYZ.proj3.tar.zip
tar -xvf XYZ.proj3.tar
cd XYZ.proj3
make
```

results in the creation of a folder `XYZ.proj3` with executable file `a.out` in it. Please make sure you do understand this. Also, inside this file there cannot be any input and/or output files and your `makefile`.