A Code Generator

- Generates target code for a sequence of three-address statements using *next-use* information
- Uses *getreg* to assign registers to variables
- For instruction $x := y$ op z
  getreg(y, z) returns a location (register) for x
- Results are kept in *registers* as long as possible:
  - Result is needed in another computation
  - Register is kept up to a procedure call or end of block
- Check if operands of three-address code are available in registers

The Code Generation Algorithm

- For each statement $x := y$ op z
  1. Set location $L = getreg(y, z)$ to get register for x
  2. If $y \notin L$ then generate
     MOV y',L
     where y' denotes one of the locations where the value of y is available (choose register if possible)
  3. Generate
     OP z',L
     where z' is one of the locations of z;
     Update register/address descriptor of x to include L
  4. If y and/or z has no next use and is stored in register, update register descriptors to remove y and/or z

Register and Address Descriptors

- A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

  `MOV a,R0` "`R0` contains `a`"

- An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

  ```
  MOV a,R0
  MOV R0,R1
  ```
  "`a` in `R0` and `R1`"

The *getreg* Algorithm

- To compute *getreg*(y, z)
  1. If y is stored in a register R and R only holds the value y, and y has no next use, then return R;
     Update address descriptor: value y no longer in R
  2. Else, return a new empty register if available
  3. Else, find an occupied register R;
     Store contents (*register spill*) by generating
     `MOV` R`,`M
     for every M in address descriptor of y;
     Return register R
  4. Return a memory location

Code Generation Example

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*
		Registers empty	
`t := a - b`	`MOV a,R0` `SUB b,R0`	`R0` contains `t`	`t` in `R0`
`u := a - c`	`MOV a,R1` `SUB c,R1`	`R0` contains `t` `R1` contains `u`	`t` in `R0` `u` in `R1`
`v := t + u`	`ADD R1,R0`	`R0` contains `v` `R1` contains `u`	`u` in `R1` `v` in `R0`
`d := v + u`	`ADD R1,R0` `MOV R0,d`	`R0` contains `d`	`d` in `R0` `d` in `R0` and memory
live(**d**)=true all other dead			