CS301 Compiler Design and Implementation **Handout # 28**
Prof. Lyn Turbak **January 6, 2004**
Wellesley College

Problem Set 5 Solutions

Note: This assignment was posted with four problems, but the fourth problem was eventually dropped. I increased the point values of the other problems to add up to 100 points.

Problem 1 [50]: ROBSTER = LOOPSTER + **Robotic Primitives**

In this problem, you were asked to extend LOOPSTER with ROBSTER's robotic primitives.

Voidops

Let's begin with the voidops:

```
and compVoidop op senv =
  match op with
    (* ... Loopster voidops omitted here ... *)
  | Wait -> gen [Xgdy; (* Move tenths to Y register *)
                 Jsr (Ext (AddrLabel "wait"))]
  | Clear -> gen [Jsr (Ext (AddrLabel "lcd-clear"))]
  | Top -> gen [Jsr (Ext (AddrLabel "lcd-top"))]
  | Bottom -> gen [Jsr (Ext (AddrLabel "lcd-bottom"))]
  | TalkTo -> gen [Jsr (Ext (AddrLabel "talk-to"))]
  | On -> gen [Jsr (Ext (AddrLabel "motors-on"))]
  | Off -> gen [Jsr (Ext (AddrLabel "motors-off"))]
  | Toggle -> gen [Jsr (Ext (AddrLabel "motors-toggle"))]
  | This -> gen [Jsr (Ext (AddrLabel "motors-this-way"))]
  | That -> gen [Jsr (Ext (AddrLabel "motors-that-way"))]
  | Rd -> gen [Jsr (Ext (AddrLabel "motors-reverse"))]
```

The `wait`, `clear`, `top`, and `bottom` operations are straightforward, since they just call prolog subroutines. The motor operations are more challenging. To simplify matters, we assume that each compiles to subroutine call. This is probably a good idea, since the subroutine body will only appear once in the compiled code rather than at every invocation of the motor operation, potentially leading to much smaller code.[1]

Manipulating the motors requires controlling an 8 bits byte: the high four bits control power (0=off, 1=on), and the low four bits control direction (0=this-way (green),1=that-way (red)). (Four bit quantities are also known as **nibbles**; there are two nibbles in a byte.) Writing such byte to address `$7000` changes the motor state. A complication is that there is no way to *read* the motor state from the HANDYBOARD. Since some of the motor manipulations implied by the motor operations require knowing the motor state, this state must be kept by the generated code. A natural way is to maintain the state in a `motors-state` pseudo-variable.

```
motors-state:                   ; High 4 bits = on/off, low 4 bits = direction
        fcb 0
```

We also need to maintain the selected set of motors (as specified by `talk-to`). Conceptually, the selection information is four bits (one for each motor) = one nibble. Since this nibble will

[1] This is not always the case, though. Including a subroutine for every motor operation can sometimes *increase* the code size if the operation is never called. It would be best to include the subroutine only if the operation is invoked at least once (perhaps even twice). Also, note that calling a subroutine incurs a run-time overhead compared with generating the operation code "in-line". This is a classic time/space tradeoff in compilation. In the case of robot controllers with small memories, space is usually at a premium, and so we tend to optimize for smaller code.

often be combined with the power (high) nibble of `motors-state` and the direction (low) nibble of `motors-state`, it is reasonable to keep two copies of the selection nibble: a copy (`motors-low`) in which the selection bits are in the low nibble of a byte (and the high nibble is 0); and a copy (`motors-high`) in which the selection nibble are at the high nibble of a byte (and the low nibble is 0):

```
motors-low:                     ; Selected motors as low nibble, 0 as high
        fcb 0
motors-high:                    ; Selected motors as high nibble, 0 as low
        fcb 0
```

In this approach, the `talk-to` subroutine sets both `motors-low` and `motors-hi`:

```
talk-to:
        cpd #15
        bhi invalid-motor-set
        stab motors-low
        aslb                    ; Perform 4 left shifts to make high
        aslb
        aslb
        aslb
        stab motors-high
        rts
invalid-motor-set:
        jsr lcd-clear
        ldx #invalid-motor-string
        jsr display-string
        jsr lcd-bottom
        jsr display-unsigned-word
        jsr wait-for-stop
        jmp main-loop           ; Evaluate again when stop pressed
invalid-motor-string:
                fjs \"Invalid motors:\"
```

Note that the `bhi` test is an *unsigned* test, and thus correctly handles any number outside the range $[0-15]$. The four left shift operations `aslb` could alternatively be expressed via `ldaa #16 mul`, but this requires 12 cycles and the four `aslb` instructions require onely 8 cycles. In other approaches, the shifts can be performed instead in the `on`, `this-way`, etc. operations, but performing them in `talk-to` usually ends up performing fewer shifts overall.

Turning motors on is straightforward — ORing the selected bits with the motor state does the trick:

```
motors-on:
        ldaa motors-high
        oraa motors-state ; Turns on selected motors; other motors unchanged
        staa motors-state
        staa motor-port
        rts
```

Since all direction bits in `motors-high` are 0, the OR operation leaves directions unchanged.

Toggling motors is similar, except we XOR the selected bits with the motor state:

```
motors-toggle:
        ldaa motors-high
        eora motors-state ; Toggles selected motors; other motors unchanged
        staa motors-state
        staa motor-port
        rts
```

Again, since all direction bits in `motors-high` are 0, the XOR operation (`eora`) leaves directions unchanged.

Turning motors off is a little bit trickier:

```
motors-off:
        ldaa motors-high
        eora #$ff         ; Invert all bits; after, all direction bits are 1
        anda motors-state ; Turns off selected motors; other motors unchanged
        staa motors-state
        staa motor-port
        rts
```

Inverting the selection bits puts a 0 in each selected bit position, and the AND operation turns off the motors at these positions. Since each unselected power bit and each direction bits in `motors-high` is 0, its inverse is 1, so the AND operation will not change the value of any bit positions other than the selected ones. It is possible to replace `eora #$ff` by `eora motors-state` without changing the behavior, but the former is more efficient than the latter (2 cycles vs. 4 cycles).

Each of the on/off operations has its analog in the direction operations, the only difference being that the low nibble of `motors-state` is manipulated rather than the high nibble.

- `motors-that-way` is like `motors-on`:

```
motors-that-way:
        ldaa motors-low
        oraa motors-state ; Sets selected motors that-way; other motors unchanged
        staa motors-state
        staa motor-port
        rts
```

- `motors-reverse-way` is like `motors-toggle`:

```
motors-reverse:
        ldaa motors-low
        eora motors-state ; Reverses selected motors; other motors unchanged
        staa motors-state
        staa motor-port
        rts
```

- `motors-this-way` is like `motors-off`:

```
motors-this-way:
        ldaa motors-low
        eora #$0f         ; Invert all bits; after, all power bits are 1
        anda motors-state ; Sets selected motors this-way; other motors unchanged
        staa motors-state
        staa motor-port
        rts
```

Valops

Now we'll consider the two valops: `sensor` and `switch`. As with the voidops, it makes sense for these to be subroutines:

```
and compValop op senv =
  match op with
    (* ... Loopster valops omitted here ... *)
  | Sensor -> gen [Jsr (Ext (AddrLabel "sensor"))]
  | Switch -> gen [Jsr (Ext (AddrLabel "switch"))]
```

The `sensor` subroutine is the more straightforward one. It involves calling `analog-read` after checking the port number is in range:

```
sensor:                         ; Sensor port should be 0--7 in B
        cpd #7
        bhi port-out-of-range
        tba                     ; Move port to A
        jsr analog-read         ; Answer in A
        tab                     ; Move result to B
        clra                    ; Clear high bits of D; result now in D
        rts
port-out-of-range:
        ldx #invalid-port-string
        jsr display-string
        jsr display-unsigned-word
        jsr wait-for-stop
        jmp main-loop           ; Evaluate again when stop pressed
invalid-port-string:
         fjs \"Invalid port:\"
```

The `switch` subroutine is much more challenging (Fig. 1), because there are many special cases:

- Any *unsigned* port number > 17 is out of range. As in `sensor`, the `bhi` instruction is the right thing here.
- Performing `switch` on an analog port must compare the analog reading to a threshold (= 127). Readings less than the threshold register as true (remember, low readings are generally "a lot" of the quantity sensed) while high readings register as false.
- The rest of the ports (7–17) are digital – i.e., they denote a single bit. But ports 7–9 come from bits 1,2, and 8 of `porta`, while ports 10–17 come from bits 1–8 of `digital-in`. How can these special cases be handled without a rat's nest of conditionals?

  Selecting between `porta` and `digital-in` can be done via a single branch, and joining the two arms of the branch (at `switch-digital-join`) avoids duplicating code. After this join, it is assumed that the appropriate byte is in register `A`.

  Now, how do we deal with the non-uniform mapping between port numbers and bit positions? What we'd really like is a function that returns the bit position given the port number. We can get this by looking up the port in an **assembly code table** that stores the bit position of port `p` at address `#port-mask-table + p`, where `#port-mask-table` is the address of the first byte in the table. In Fig. 1, the first seven bytes of the table are never used and are arbitrarily set to 0. We can avoid storing these seven bytes if we index the table at `#port-mask-table + p + 7` instead of at `#port-mask-table + p`. Again we have a tradeoff: are seven bytes of space worth several cycles to perform an addition every time `switch` is called? Since seven bytes is very small, I have made the tradeoff to reduce run-time, but the other option is reasonable, too.

```
switch:                         ; Switch port should be 0--17 in B
        cpd #17
        bhi port-out-of-range
        cmpb #6                 ; Switches 0-6 are analog sensor ports
        bhi switch-digital
switch-analog:                  ; Analog ports (0-255) handled specially
        tba                     ; Move port in B to A
        jsr analog-read         ; Analog answer in A; need to compare to 127
        anda #$80               ; Mask high bit: if = 1, > 128; else <= 127
        eora #$80               ; Invert high bit to get result
        clrb                    ; Clear bits in B; now D contains bool result
        rts
switch-digital:                 ; Digital port 7-17 in B
        cmpb #9                 ; Do we read from PORTA or DIGITAL-IN?
        bhi switch-digital-digital-in
switch-digital-porta:
        ldaa porta
        bra switch-digital-join
switch-digital-digital-in:
        ldaa digital-in
switch-digital-join:            ; Sensed byte now in A; port in B
        ldx #port-mask-table    ; Load address of mask table into X
        abx                     ; Add B to X, which now has mask address
        anda 0,X                ; Masked switch now in A
        eora 0,X                ; Invert switch bit
        clrb                    ; Clear B bits; now D contains bool result
        rts
port-mask-table:
        fcb 0    ; switch 0
        fcb 0    ; switch 1
        fcb 0    ; switch 2
        fcb 0    ; switch 3
        fcb 0    ; switch 4
        fcb 0    ; switch 5
        fcb 0    ; switch 6
        fcb $01 ; switch 7
        fcb $02 ; switch 8
        fcb $80 ; switch 9
        fcb $01 ; switch 10
        fcb $02 ; switch 11
        fcb $04 ; switch 12
        fcb $08 ; switch 13
        fcb $10 ; switch 14
        fcb $20 ; switch 15
        fcb $40 ; switch 16 = STOP
        fcb $80 ; switch 17 = START
```

Figure 1: Implementation of the `switch` subroutine.

The clever code for returning booleans after `jsr analog-read` and `abx` is due to Amrutha Nagarajan and Jue Wang. This code takes advantage of the fact that any non-zero value counts as true in ROBSTER. There are many other correct, but less efficient, strategies. For example, the instructions

```
        anda #$80 ; Mask high bit: if = 1, > 128; else <= 127
        eora #$80 ; Invert high bit to get result
        clrb      ; Clear bits in B; now D contains bool result
        rts
```

could be replaced by

```
        cmpa #$80               ; 128 is the threshold:
        bhs switch-analog-false ; >= 128 is considered false
switch-analog-true:             ; <= 127 is considered true
        ldd #1
        rts
switch-analog-false:
        ldd #0
        rts
```

Excluding the common `rts` instruction, the former instruction sequence costs 6 cycles while the latter costs 7 cycles (and is longer in code size).

Problem 2 [25]: Exit and continue

Conceptually, the `exit` and `continue` constructs manipulate the test and done labels of the innermost enclosing `while` loop: `exit` jumps to the done label while `continue` jumps to the test label. To implement these constructs in the ROBSTER compiler, we adopt the following strategy:

- extend the `compStm` function to pass the "current" `while` test and done labels as extra information to every recursive call of `compStm`. This way, each occurence of `exit` or `continue` in a `while` body will "see" the correct labels.

  There are many ways to pass the label information. We choose to pass them as a pair of names (strings) called `labs`:

  ```
  and compStm stm senv labs = ...
  (* labs is a new argument that denotes a pair of
      while loop test/done labels *)
  ```

- Change the top-level call to `compStm` in `compile` to pass "default" label information indicating that the current statement is not inside a loop. We use the string pair `("","")` for this purpose:

  ```
  let rec compile (Pgm(fmls,body)) =
    let senv0 = SEnv.make fmls
     in
       seq (glue [beginCode fmls senv0;
                  compStm body senv0 ("","");
                  endCode ()])
  ```

- change the `while` clause of `compStm` to pass the test and done labels of the `while` loop to the call of `compStm` on the loop body (see the line marked `(***)` below):

```
| While(test,body) ->
    (* while loop that executes body stm as long as test exp is true *)
    let testLabel = newLabel "whileTest"
    and bodyLabel = newLabel "whileBody"
    and doneLabel = newLabel "whileDone"
      in glue [
           gen [Label testLabel];
                compExp test senv;
                (* Convention: 0 is false, anything else is true *)
                gen [Cpd (Imm (ImmWord 0));
                     Bne (BraLabel bodyLabel);
                     Jmp (Ext (AddrLabel doneLabel));
                     Label bodyLabel]; (* Need to jump in case bodyInstrs is large *)
                compStm body senv (testLabel, doneLabel);
                gen [Jmp (Ext (AddrLabel testLabel));
                     Label doneLabel]
         ]
```

- Add a clause to `compStm` that handles `exit` by jumping to the done label. When the empty string label is encountered, it means the `exit` is not within a `while` loop, and an error is signalled:

```
| Exit -> (match labs with
            (_,doneLabel) ->
              if doneLabel = "" then
                raise (CompError "exit not within a while loop")
              else
                gen [Jmp (Ext (AddrLabel doneLabel))])
```

- Add a clause to `compStm` that handles `continue` by jumping to the test label. When the empty string label is encountered, it means the `continue` is not within a `while` loop, and an error is signalled:

```
| Cont -> (match labs with
            (testLabel,_) ->
              if testLabel = "" then
                raise (CompError "exit not within a while loop")
              else
                gen [Jmp (Ext (AddrLabel testLabel))])
```

- Change all other clauses of `compStm` to pass `labs` unchanged to any recursive calls of `compStm`.

Problem 3 [25]: Desugaring

In this problem, you were asked to implement five new ROBSTER desugarings by extending the `desugarRules` function. Below, we give desugaring rules and OCAML `desugarRules` code for each desugaring:

1. `loop`: The desugaring rule for `loop` is

   `(loop` S_{body}`)` $\Rightarrow$ `(while true` S_{body}`)`,

   which can be expressed via the following `desugarRules` clause:

```
| S.Seq [Sym "loop"; stm] -> S.Seq [Sym "while"; Sym "true"; stm]
```

2. `wait-until`: The desugaring rule for `wait-until` is

```
(wait-until E) ⇒
  (seq (while E (skip))
       {Reach here when E is false.}
       (while (! E) (skip))
       {Reach here when E is true (after having been false).}
       ),
```

   which can be expressed via the following `desugarRules` clause:

```
| S.Seq [Sym "wait-until"; exp] ->
    S.Seq [Sym "seq";
           S.Seq [Sym "while"; exp; S.Seq [Sym "skip"]];
           S.Seq [Sym "while"; S.Seq [Sym "!"; exp]; S.Seq [Sym "skip"]]]
```

3. `++`: The desugaring rule for `++` is

```
(++ I ⇒ (<- I (+ I 1))),
```

   which can be expressed via the following `desugarRules` clause:

```
| S.Seq [Sym "++"; Sym var] ->
    S.Seq [Sym "<-"; Sym var; S.Seq [Sym "+"; Sym var; Int 1]]
```

4. `+=`: The desugaring rule for `+=` is

```
(+= I E) ⇒ (<- I (+ I E)),
```

   which can be expressed via the following `desugarRules` clause:

```
| S.Seq [Sym "+="; Sym var; exp] ->
    S.Seq [Sym "<-"; Sym var; S.Seq [Sym "+"; Sym var; exp]]
```

5. `for`: The desugaring rule for `for` is

```
(for I E_lo E_hi S_body) ⇒
  (decl I E_lo
    (decl I_hi E_hi {I_hi is fresh}
      (while (< I I_hi)
        (seq E_body (++ I))))),
```

   which can be expressed via the following `desugarRules` clause:

```
| S.Seq [Sym "for"; Sym var; loExp; hiExp; body] ->
    let hiVar = freshId ()
     in S.Seq [Sym "decl"; Sym var; loExp;
               S.Seq [Sym "decl"; Sym hiVar; hiExp;
                      S.Seq [Sym "while";
                             S.Seq [Sym "<"; Sym var; Sym hiVar];
                             S.Seq [Sym "seq";
                                    body;
                                    S.Seq [Sym "++"; Sym var]]]]]
```