CS301 Compiler Design and Implementation **Handout # 26**
Prof. Lyn Turbak **January 6, 2004**
Wellesley College

Problem Set 3 Solutions

Problem 1 [50]: A Condex to PostFix Translator

Translation

A working Condex to PostFix translator is shown in Figs. 1–2. The key to `transPgm` is defininining an initial `argMap` function that translates Condex argument indices into offsets in the PostFix stack. The fact that PostFix arguments appear in reverse order on the stack and used 0-based indexing rather than 1-based indexing leads to the formula `n - i`, where `n` is the number of parameters to the program and `i` is the index of the Condex argument.

The `transExps` function effectively maps `transExp` over the given expression list and appends the resulting PostFix commands together. But since each expression evaluation pushes a new value on the stack, the argument map `amap` must be updated to account for this fact. This is accomplished with the `push` function, which increments the PostFix stack position of of every Condex argument index.

The `transExp` function dispatches on the kind of Condex expression. Literals are straightforward to handle, and the presence of the `amap` argument makes arguments references straightforward as well. Branches translate into a selection between two executable sequences (one for each branch), only one of which is executed. Wrapping the branches in executatble sequences is essential for preserving the semantics that only one branch is executed.

The most complex case for `transExp` is handling primitive operator applications. Code is generated for pushed the arguments of the application on the stack, and `transPrimop` is responsible for generating the code that performs the application. Most Condex primitives correspond directly to PostFix primtives. The interesting cases are `Not`, `And`, and `Or`. The tricky aspect of handling these is that "true" can be represented as any non-zero number. There are many correct PostFix command sequences for these operations; Fig. 2 gives some particularly concise solutions. Here are a few other valid solutions:

- `Not1: [P.Int(0); P.Int(1); P.Sel]`
- `And1: [P.Int(1); P.Int(0); P.Sel; P.Int(0); P.Sel]`
  `And2: [P.Int(0); P.NE; P.Swap; P.Int(0); P.NE; P.Mul]`
- `Or1: [P.Int(1); P.Int(0); P.Sel; P.Int(1); P.Swap; P.Sel]`
  `Or2: [P.Int(0); P.NE; P.Swap; P.Int(0); P.NE; P.Add; P.Int(0); P.NE]`

`And2` and `Or2` are interesting in the sense that the make use of arithmetic properties of `0` and `1` to perform boolean opertions. They suggest some very concise alternative solutions that unfortunately are incorrect:

- `And4-wrong: [P.Mul; P.Int(0); P.NE]`. This fails because with finite integers can get 0 in other ways. E.g., with 4 bit ints, `[8; 2; mul]` is 0. In OCaml, which has 31 bit ints, `[32768;65536;mul]` is 0. But `(& 8 2)` and `(& 32768 65536)` should be 1!
- `Or4-wrong: [P.Add; P.Int(0); P.NE]`. This fails because can also get 0 by adding an integer to its additive inverse. E.g.`[-1; 1; add]` is 0, but `(+ -1 1)` should be 1.

```
module CondexToPostFix :

  sig
    exception TransError of string
    val transPgm : Condex.pgm -> PostFix.pgm
    val transExp : Condex.exp -> (int -> int) -> PostFix.com list
    val transExps : Condex.exp list -> (int -> int) -> PostFix.com list
  end

=

struct

  exception TransError of string

  (* Handy abbreviations *)
  module C = Condex
  module P = PostFix

  let push amap = fun i -> (amap i)+1

  let rec transPgm (C.Pgm(n,body)) =
    let argMap i =
      if (i <= 0) || (i > n) then
        raise (TransError ("Illegal arg index: "
                           ^ (string_of_int i)))
      else
        n - i (* account for fact that args are reversed on stack *)
     in
       P.Pgm(n, transExp body argMap)

  (* Translate Condex expression [e1;e2;...;en] into a sequence of
     PostFix commands that, when executed on a stack s
     will yield a stack (vn::...::v2::v1::s), where vi is the value of ei.
     The amap argument tracks the index of each Condex program argument
     on the stack. *)
  and transExps exps amap =
    match exps with
       [] -> []
    | e::es -> (transExp e amap) @ (transExps es (push amap))
```

Figure 1: Condex to PostFix translator, Part 1.

```
(* Translate Condex expression exp into a sequence of
     PostFix commands that, when executed on a stack s
     will yield a stack (v::s), where v is the value of exp.
     The amap argument tracks the index of each Condex program argument
     on the stack. *)
  and transExp exp amap =
    match exp with
      C.Lit n -> [P.Int n]
    | C.Arg i -> [P.Int (amap i); P.Get]
    | C.Bra (test,con,alt) ->
         (transExp test amap)
       @ [P.Seq (transExp con amap)]
       @ [P.Seq (transExp alt amap)]
       @ [P.Sel;P.Exec]
    | C.PrimApp (op, rands) -> (transExps rands amap) @ (transPrimop op)

  and transPrimop op =
    match op with
      C.Add -> [P.Add]
    | C.Sub -> [P.Sub]
    | C.Mul -> [P.Mul]
    | C.Div -> [P.Div]
    | C.Rem -> [P.Rem]
    | C.LT -> [P.LT]
    | C.EQ -> [P.EQ]
    | C.GT -> [P.GT]
    | C.Not -> [P.Int(0); P.EQ]
    | C.And -> [P.Int(0); P.NE; (* convert 2nd arg to 0/1 *)
                P.Int(0); P.Sel (* use 1st arg to choose result *)
               ]
    | C.Or -> [P.Int(0); P.NE;  (* convert 2nd arg to 0/1 *)
               P.Int(1); P.Swap; P.Sel (* use 1st arg to choose result *)
              ]

end
```

Figure 2: CONDEX to POSTFIX translator, Part 2.

Testing

A complete testing program for the Condex to PostFix translator is shown in Fig. 3. The tricky aspect of testing is making the output of executing the resulting PostFix program (an instance of `PostFixInterp.ans`) compatible with the expected result in the Condex test suite (an instance of `CondexInterpTest.result`). Fortunately, there is a straightforward relationship between the two: each `PostFixInterp.IntAns(i)` corresponds to `CondexInterpTest.Val(i)`, and each `CondexInterpTest.Err(s)` corresponds to `PostFixInterp.ErrAns(s)`.

There are many ways to address this incompatibility; here are two:

1. Transform the result of running the PostFix interpreter (an instance of `PostFixInterp.ans`) into a instance of running the Condex interpreter (`CondexInterpTest.result`). This is the approach taken in Fig. 3, where the `ans2Result` function performs this translation.

2. Transform the Condex test suite (where expected results are of type `CondexInterpTest.result`) into a test suite where the expected results are of type `PostFixInterp.ans`. Doing this by hand would be tedious, but it would not be difficult to write an OCaml function that automatically performed the tranformation via something like an inverse to `ans2Result`.

```
module CondexToPostFixTest = struct

  let ans2Result ans =
    match ans with
      PostFixInterp.IntAns(i) -> CondexInterpTest.Val(i)
    | PostFixInterp.ErrAns(s) -> CondexInterpTest.Err(s)

  module TransTester =
    MakeTester (
      struct
        type prog = string
        type arg = int
        type res = CondexInterpTest.result

        let trial progString args =
          try
            ans2Result
              (PostFixInterp.run
                (CondexToPostFix.transPgm
                  (Condex.string2Pgm progString))
               args)
            with
              Condex.SyntaxError(str) -> CondexInterpTest.Err(str)
            | CondexToPostFix.TransError(str) ->  CondexInterpTest.Err(str)
        let arg2String = string_of_int
        let resEqual = (=)
        let res2String = CondexInterpTest.result2String
      end
    )

  let test () = TransTester.testEntries CondexInterpTest.entries

  let trans s = PostFix.pgm2String (CondexToPostFix.transPgm (Condex.string2Pgm s));;

end
```

Figure 3: A complete testing program for the CONDEX to POSTFIX translator.

Problem 2 [50]: 6811 Programming

1. [15] **gcd** One way to calculate the GCD of two unsigned 16-bit numbers is presented in Fig. 4. This solution also includes debugging code that displays the values of `a` and `b` at the beginning of every iteration of the loop. The assembly code comments (here and in other problems) are crucial for understanding the code and its invariants.

   Note that pseudo-registers (other than `prompt` and `wordread`) are not necessary in this problem. Only registers `D` and `X` and one stack slot are necessary (not even `Y` is needed). Some student solutions swapped the meanings of `a` and `b`; since `gcd(a,b)` = `gcd(b,a)`, this can still give the right answer, even if the code is "wrong".

2. [15] **follow-light** One approach to the light-following problem is presented in Fig. 5. This solution uses the *difference* between the left and right photocell sensors to determine which way the SciBorg should turn. This is more robust than comparing the individual sensors to a threshold. Note that only the `A` and `B` registers are needed; the `tab` and `cba` instructions are very useful in this regard.

   Several teams got confused and programmed the robot to turn *away* from the light. There were several sources of confusion: (1) the fact that high photocell values indicate low amounts of light; some teams assumed the opposite; (2) turning on the left motor turns the SciBorg *right*, not left; (3) some teams were confused which sensors were attached to which sensor ports and which motors were attached to which motor ports.

3. [20] **display-binary** One approach to displaying the binary representation of an unsigned 16-bit number is presented in Fig. 6 The key to this solution is using recursion (in conjunction with the HandyBoard stack) to display the bits on the *way out* of the recursion.

   The code in Fig. 6 is clever and concise, combining the best ideas from my original solution and student solutions. The `lsrd` instruction is a *much* cheaper way to divide by 2 than using `idiv` (3 cycles vs. 41!), and remainder by 2 can be calculated via `andb #1` (2 cycles). Performing `andb #1` *after* the recursive call means that `D` is not altered before the call and only register `B` (not all of `D`) needs to be saved across the call. The fall-through before `db-base-case:` prevents duplicating two lines of assembly code. Calling the prolog routine `display-bit` (after `cmpb #0`) is cheaper than calling `display-unsigned-byte-b` or (even more expensive) `display-unsigned-word`. The tail call `jmp display-bit` is an optimization over the sequence `jsr display-bit rts`.

   Isn't hacking assembly code fun?

```
;;; -----------------------------------------------------------------------
;;; GCD
;;; Prompts the user for two unsigned 16-bit numbers in the top row of the LCD
;;; and display the GCD of these in the bottom row. The result should also be
;;; returned in the D register. Returns to main menu when STOP button is pressed.
gcd:
        ldd #gcd-a                ; Read input A from user
        std prompt
        jsr read-unsigned-word
        ldd wordread              ; D holds A.
        xgdx                      ; X now holds A, D is garbage
        ldd #gcd-b                ; Read input B from user
        std prompt
        jsr read-unsigned-word
        ldd wordread              ; Now D holds B and X holds A
        xgdx                      ; Now D holds A and X holds B
gcd-loop:                         ; Invariant: D holds A,  X holds B.
        jsr lcd-clear             ; Begin debugging code
        pshx                      ; Save X (B), since will be overwritten
        ldx #gcd-a                ; Display "a="
        jsr display-string
        jsr display-equal
        jsr display-unsigned-word ; Display A
        jsr display-space         ; Display " b="
        ldx #gcd-b
        jsr display-string
        jsr display-equal
        pulx                      ; Restore X (B), since will be overwritten
        xgdx                      ; Now D holds B and X holds A
        jsr display-unsigned-word ; Display B
        xgdx                      ; Now D holds A and X holds B
        ldy #20                   ; Wait for 2 seconds
        jsr wait                  ; End debugging code
        cpx #0                    ; Base case: is A=0?
        beq gcd-done              ; If so, we're done!
        pshx                      ; Save B
        idiv                      ; D = A mod B (overwrites X)
        pulx                      ; Restore B in X
        xgdx                      ; Now D has B and X has A mod B
        bra gcd-loop              ; Deja vu all over again!
gcd-done:
        jsr lcd-bottom
        jsr display-unsigned-word ; Displays contents of D (= A)
        jmp wait-for-stop         ; Tail call to wait-for-stop
gcd-a:
        fjs "a"
gcd-b:
        fjs "b"
```

Figure 4: 6811 assembly code for calculating the GCD of two unsigned 16-bit numbers.

```
;;; -----------------------------------------------------------------------
;;; FOLLOW-LIGHT
;;; Causes SciBorg to follow a light using the two photosensors at the
;;; front of the vehicle. Use the difference between the photosensor readings
;;; to determine which motor is on. Stop when STOP button pressed or when
;;; either bumper is pressed.
follow-light:
        ldx #digital-in
        brclr 0,X $40 flt-done       ; Done when STOP pressed
        ldx #porta                   ; Switch 7 @ bit 0; switch 8 @ bit 1
        brclr 0,X $01 flt-done             ; Done when front bumper pressed
        brclr 0,X $02 flt-done             ; Done when back bumper pressed
        ldaa #2                      ; Read left photocell
        jsr analog-read
        jsr lcd-clear                ; Debugging code: display left reading
        jsr display-unsigned-byte-a
        jsr display-space
        tab                             ; Move left reading to b
        ldaa #3                      ; Read right photocell
        jsr analog-read              ; and store in A
        jsr display-unsigned-byte-a  ; Debugging code: display right reading
        jsr wait-100msec
        cba                             ; Set CCR based on A-B
                                     ; is right greater than left?
                                     ; i.e. does right have *less* light than left?
        bhi flt-gt                   ; If yes, branch to FLT-GT
        ldaa #$10                    ; Else turn right (to increase left light)
        staa motor-port
        bra follow-light             ; Repeat
flt-gt:
        ldaa #$20                    ; Turn left (to increase right light)
        staa motor-port
        bra follow-light             ; Repeat
flt-done:
        ldaa #$00                    ; Stop motors
        staa motor-port
        rts
```

Figure 5: 6811 assembly code that causes a SciBorg to follow a flashlight.

```
;;; ----------------------------------------------------------------------
;;; DISPLAY-BINARY
;;; Prompts the user for an unsigned 16-bit number on the top row of the LCD
;;; and displays the binary representation of this number (without leading zeroes)
;;; in the bottom row. Returns to main menu when STOP button is pressed.
display-binary:
        ldd #db-prompt                ; Read input N from user
        std prompt
        jsr read-word
        ldd wordread                  ; D holds N
        jsr lcd-bottom
        jsr dispbin                   ; Call recursive function
        jmp wait-for-stop             ; Tail call to wait-for-stop
;;; DISBIN: displays in binary the value in the D register,
;;; using a recursive algorithm
dispbin:
        cpd #1                        ; Base case: N <= 0
        bls db-base-case
        pshb                          ; Save lower byte, which has N mod 2 in it
        lsrd                          ; Divide by 2 (*much*) cheaper than IDIV
        jsr dispbin                   ; Display all but last bit
        pulb                          ; Restore lower byte
        andb #1                       ; Mask lowest order bit
                                      ; And fall through to display
db-base-case:
        cmpb #0
        jmp display-bit               ; Tail call to prolog routine
db-prompt:
        fjs "n"
```

Figure 6: 6811 assembly code for displaying the binary representation of an unsigned 16-bit number.