CS301 Compiler Design and Implementation **Handout # 27**
Prof. Lyn Turbak **January 7, 2004**
Wellesley College

Problem Set 4 Solutions

Problem 1 [100]: A PostFix to 6811 Translator

You were asked to write a PostFix to 6811 translator from scratch. Here we present one possible solution in stages.

A Code Abstraction

To abstract over the process of code generation, it is helpful to have an abstraction for gluing code fragments together. Such an abstraction makes it easy to experiment with different strategies, such as storing compiled subroutines "in-line" vs. storing compiled subroutines in a different part of memory. Fig. 1 presents the signature for a code abstraction we will use. Code fragments are created by `gen`, `genStr`, and `genSubr` functions; they are glued together by `glue`, and they are converted into 6811 instructions by `seq`.

`type code` The abstract type of the code fragements being generated.

`val gen: Instr68HC11.instr list -> code`
Turn a "main" 6811 code sequence into a code fragment.

`val genSubr: code -> (Instr68HC11.label -> code) -> code`
Given the code fragment c for a subroutine body and a function f that maps a label to code, returns a code fragment that abstractly contains 2 parts:

1. the code c labeled with new label l.
2. The code that results from invoking f on l.

`val genStr: string -> (Instr68HC11.label -> code) -> code`
Given a string s and a function f that maps a label to code, returns a code fragment that abstractly contains 2 parts:

1. A sequence of bytes labeled with a new label l that begins with a length byte and is followed by the characters of s. It is an error if s has more than 255 characters.
2. The code that results from invoking f on l.

`val glue: code list -> code`
Returns the result of gluing together a list of code fragments.

`val seq: code -> Instr68HC11.instr list`
Converts a code fragment into executable 6811 code.

Figure 1: Signature of the code abstraction.

The simplest implementation of the code abstraction is shown in Fig. 2. Here the `code` type is simply a list of 6811 instructions. Both `genSubr` and `genStr` are implemented in terms of a `gen2` function that has its first (i.e., main) instruction sequence "jump around" the second instruction sequence. It is assumed that the second instruction sequence could be anywhere in memory; it just happens to be plunked down in the middle of the main instruction sequence. We assume the existence of a `newLabel` function that generates fresh labels. Here is one such function:

```
let newLabel = StringUtils.genFresh "_"
```

```
type code = Instr68HC11.instr list
let gen instrs = instrs

let gen2 (instrs1, instrs2) =
  let endLabel = newLabel "end"
   in   instrs1
        @ [Jmp (Ext (AddrLabel endLabel))] (* Jump around instrs2 *)
           (* Use jump rather than branch since don't know how big *)
            (* Could be cleverer by calculating size! *)
        @ instrs2
        @ [Label endLabel]

let genSubr subrCode fcn =
  let subrBegin = newLabel "subr_begin"
   in gen2 (fcn subrBegin, [Label subrBegin] @ subrCode)

let genStr str fcn =
  let strBegin = newLabel "string_begin"
   in gen2 (fcn strBegin, [Label strBegin; FormJavaString str])

let glue instrss = List.concat instrss

let seq instrs =
  instrs @ [FormBytes [0;0;0]; (* Add a few extra byte to protect code *)
                               (* in lieu of alignment *)
            Label "pfstkmax"] (* Max range of PostFix stack *)
```

Figure 2: A simple representation of code as a list of instructions.

A fancier implementation of the code abstraction is shown in Fig. 3. Here the `code` type is a *triple* of 6811 instruction lists: (1) the main code; (2) subroutine code; and (3) string constants. In this representation, the `glue` function performs an elementwise concatenation of the sequence components of a list of triples. The `gen` function adds code to the first component, `genSubr c f` glues the code that results from calling `f` together with subroutine code c in the subroutine component, and `genStr s f` glues the code that results from calling `f` together with labeled string bytes in the string component.

One last utility we will need is a way of generating temporary addresses:

```
(* Address of temp pseudo-register (temp + i) *)
let tempAddress i temp = AddAddr (AddrLabel "temps", Addr (temp + i))
```

The Main Compiler

The main compiler code is presented in Fig. 4. The code `begin` code precedes the compiled body of the given program. Note how that body is given the label `postfix-code` and is treated as a subroutine call. This simplifies the handling of tail recursion, since every command sequence compiled by `compComs` can be assumed to end in an `rts` in a naive compilation.

In Fig. 4, *assembly code string* stands for a string defining many assembly code subroutines. These are presented in Figs. 5–8. These subroutines are a good way to avoid including common code idioms in the compiled body code. In some cases, these subroutines are not as efficient as they could be. For instance, Ivana and Mirena had much cleverer versions of `need1`, `need2`, and

```
let gen instrs = (instrs, [], []) (* Generate main code *)

let genSubr subrCode fcn =
  let subrBegin = newLabel "subr_begin"
  and (mains,subrs,strs) = subrCode
   in glue [fcn subrBegin;
            ([], [Label subrBegin] @ mains @ subrs, strs)]

let genStr str fcn =
  let strBegin = newLabel "string_begin"
   in glue [fcn strBegin;
            ([], [], [Label strBegin; FormJavaString str])]

let glue triples =
  let (mainss, subrss, strss) = ListUtils.unzip3 triples
   in (List.concat mainss, List.concat subrss, List.concat strss)

let seq (mains, subrs, strs) =
    mains @ subrs @ strs
  @ [FormBytes [0;0;0]; (* Add a few extra byte to protect code in lieu of alignment *)
     Label "pfstkmax"] (* Max range of PostFix stack *)
```

Figure 3: A fancy representation of code as a triple of instruction lists: (1) main code (2) strings and (3) subroutines.

`need3` that used address arithmetic rather than `inx` and `dex`.

```
let rec compile (Pgm(n,coms)) =
    seq (glue [beginCode n; compComs coms])

  and beginCode n =  (* Standard boilerplate for beginning of program *)
   glue
   [gen
    (Parser.parseString  assembly code string);
     glue (List.map
            (fun i ->
              gen
              [ Label ("read-arg" ^ (string_of_int i));
                Ldd (Imm (ImmLabel ("arg" ^ (string_of_int i) ^ "-string")));
                Std (Ext (AddrLabel "prompt"));
                Jsr (Ext (AddrLabel "read-signed-word"));
                Ldd (Ext (AddrLabel "wordread"));
                Jsr (Ext (AddrLabel "pushd"));
                Rts;
                Label ("arg" ^ (string_of_int i) ^ "-string");
                FormJavaString ("arg" ^ (string_of_int i))
              ] )
            (ListUtils.fromTo 1 n));
    gen (Parser.parseString
          "main-loop:
           lds #$ffff                  ; Reset control stack ptr
           ldx #pfstkmin               ; Store initial stack ptr in X
          ");
    glue (List.map (fun i ->
                    gen [Jsr (Ext (AddrLabel ("read-arg" ^ (string_of_int i))))])
                   (ListUtils.fromTo 1 n));
    gen (Parser.parseString
          "        jsr lcd-clear
                   jsr postfix-code    ; Invoke subroutine for body of PostFix program
                   jsr popd
                   jsr lcd-bottom
                   ldx #result-string
                   jsr display-string
                   jsr display-equal
                   jsr display-signed-word
                   jsr wait-for-stop
                   jmp main-loop       ; Evaluate again when stop pressed
                                       ; Use jmp rather than bra because don't
                                       ; know how long code is
           postfix-code:               ; Code for body of PostFix program goes here
           ")
     ]
```

Figure 4: The main compiler code. The part labeled *assembly code string* is presented later.

```
"include "./hc11/prolog.asm" ; Include standard libraries
        pfstkmin equ $bf00          ; Bottom of PostFix stack
        main:                       ; Start of user program
                lds #$ffff          ; Put stack at end of memory
                jsr init            ; Perform initializations
                jmp main-loop       ; Start main loop
        div0-error:
                jsr lcd-clear
                ldx #div0-string
                jsr display-string
                jsr display-signed-word
                jsr wait-for-stop
                jmp main-loop           ; Evaluate again when stop pressed
        rem0-error:
                jsr lcd-clear
                ldx #rem0-string
                jsr display-string
                jsr display-signed-word
                jsr wait-for-stop
                jmp main-loop           ; Evaluate again when stop pressed
        put-neg-index:
                jsr lcd-clear
                ldx #put-neg-string
                jsr display-string
                jsr display-signed-word
                jsr wait-for-stop
                jmp main-loop
        get-neg-index:
                jsr lcd-clear
                ldx #get-neg-string
                jsr display-string
                jsr display-signed-word
                jsr wait-for-stop
                jmp main-loop
        pfstkfull:
                jsr lcd-clear
                ldx #full-string
                jsr display-string
                jsr wait-for-stop
                jmp main-loop
        cstkfull:
                jsr lcd-clear
                ldx #cfull-string
                jsr display-string
                jsr wait-for-stop
                jmp main-loop
```

Figure 5: Assembly Code, Part 1

```
pushd:                          ; Push D onto PostFix stack
                cpx #pfstkmax
                blo pfstkfull
                std 0,X
                dex
                dex
                rts
        check-bounds:               ; Check if stack reference in D is too low
                                     ; Use by put and get ops
                cpd #pfstkmin
                bhi pfstkempty         ; Jump to too-small-stack error if out of bounds
                rts
        popd:
                inx
                inx
                cpx #pfstkmin
                bhi pfstkempty
                ldd 0,X
                rts
        pfstkempty:
               jsr lcd-clear
               ldx #empty-string
               jsr display-string
               jsr wait-for-stop
               jmp main-loop
```

Figure 6: Assembly Code, Part 2

Compiling Command Sequences

The function `compComs` for compiling command sequences is shown below:

```
(* (compComs exp) compiles command list coms into a sequence of 6811
   instructions that has the effect of changing the run-time in the
   way that executing the commands would *)
and compComs coms =
  match coms with
    [] -> gen [Rts]
  | [Exec] -> gen [Jmp (Ext (AddrLabel "exec"))]
    (* Tail call for exec handled specially! *)
  | (c::cs) -> glue [compCom c; compComs cs]
```

It is assumed that every command sequence ends in a return; the top level program must be embedded in a subroutine for this to make sense. Tail recursion is implemented by handling a sequence ending in `exec` as a `jmp` rather than as a `jsr` followed by an `rts`.

This is not the only way to handle tail recursion. An alternative is to first generate `jsr`/`rts` pairs, and then remove them by a separate peephole optimization pass.

```
;; Note: have to carefully order pfstkempty relative to those
        ;; subroutines that branch to it to guarantee that all branches are
        ;; within indices -128 t0 127!
         need1:                          ; Call this when need 1 val on stack.
                                         ; Verifies that there is 1 val
                                         ; and leaves PFSTK pointing to val
                 inx
                 inx
                 cpx #pfstkmin
                 bhi pfstkempty
                 rts

        need2:                           ; Call this when need 2 vals on stack.
                                         ; Verifies that there are 2 vals
                                         ; and leaves PFSTK pointing at top val
                 inx
                 inx
                 inx
                 inx
                 cpx #pfstkmin
                 bhi pfstkempty
                 dex
                 dex
                 rts
        need3:                           ; Call this when need 3 vals on stack.
                                         ; Verifies that there are 3 vals
                                         ; and leaves PFSTK pointing at top val
                 inx
                 inx
                 inx
                 inx
                 inx
                 inx
                 cpx #pfstkmin
                 bhi pfstkempty
                 dex
                 dex
                 dex
                 dex
                 rts
```

Figure 7: Assembly Code, Part 3

```
exec:
         sts 0,X               ; Test control stack pointer
         ldy #$c100            ; to make sure it's not too close to reset vector.
         cpy 0,X

         bhi cstkfull
         jsr need1
         ldy 0,X               ; Beware: Jsr jumps to effective address,
                               ; not *contents* of effective address.
         jmp 0,Y               ; tail optimization
result-string:
         fjs ŕesult¨
div0-string:
         fjs D̈iv by 0:¨
get-neg-string:
         fjs G̈et index:¨
put-neg-string:
         fjs P̈ut index:¨
rem0-string:
         fjs R̈em by 0:¨
empty-string:
         fjs Ëmpty pstack¨
full-string:
         fjs F̈ull pstack¨
cfull-string:
         fjs F̈ull cstack¨
");
```

Figure 8: Assembly Code, Part 4

Compiling Commands

The core of the compiler is the code generation for individual PostFix commands. One approach is shown in Figs. 9–13. These code generators were designed to emphasize the efficiency of the generated code over the readability of the compiler. For instance, `swap` could be compiled via a sequence of calls to pop and push subroutines, but the generated code given here is much more efficient. The handling of `mul` in Fig. 12 is particularly tricky. In Fig. 13, note how `compRel` abstracts over the common pattern of compiling relational operations.

```
(* (compCom exp) compiles command com into a sequence of 6811
     instructions that has the effect of changing the run-time in the
     way that executing the command would *)
  and compCom com =
    match com with
      Int i -> glue [compInt i; gen [Jsr (Ext (AddrLabel "pushd"))]]
    | Str s -> genStr s
                 (fun strLabel ->
                    gen [Ldd (Imm (ImmLabel strLabel));
                         Jsr (Ext (AddrLabel "pushd"))])  (* Push string ptr on pfstk *)
    | Seq coms ->
        genSubr (compComs coms)
          (fun subrLabel ->
             gen [Ldd (Imm (ImmLabel subrLabel));
                  Jsr (Ext (AddrLabel "pushd"))]) (* Push subroutine ptr on pfstk *)
    | Exec -> gen [Jsr (Ext (AddrLabel "exec"))] (* Non-tail call for exec *)
    | Pop -> gen [Jsr (Ext (AddrLabel "popd"))]
    | Swap ->
        gen[Jsr (Ext (AddrLabel "need2"));
            Ldd (Index(0,X)); Xgdy;
            Ldd (Index(2,X)); Xgdy;
            Std (Index(2,X)); Xgdy;
            Std (Index(0,X)); Dex;
            Dex]
    | Prs ->
        gen [(* Jsr (Ext (AddrLabel "popd"));
              Pshx;
              Xgdx; *)
              Jsr (Ext (AddrLabel "need1"));
              Pshx;
              Ldx (Index(0,X));
              Jsr (Ext (AddrLabel "display-string"));
              Jsr (Ext (AddrLabel "wait-for-start-stop"));
              (* Jsr (Ext (AddrLabel "wait-1msec")); (* Give it time to display *) *)
              Pulx]
    | Pri -> gen
             [Jsr (Ext (AddrLabel "popd"));
              Jsr (Ext (AddrLabel "display-signed-word"));
              (* Jsr (Ext (AddrLabel "wait-1msec")) (* Give it time to display *) *)
              Jsr (Ext (AddrLabel "wait-for-start-stop"))]
```

Figure 9: Compiling PostFix commands, part 1.

```
| Sel ->
   let selTrue = newLabel "sel-true"
   and selJoin = newLabel "sel-join"
    in gen
        [Jsr (Ext (AddrLabel "need3")); (* Ensure we at least have an index;
                                           leaves PFSP pointing at alternate *)
         Ldd (Index(4,X)); (* Load test into D *)
         Bne (BraLabel selTrue);
         Ldd (Index(0,X)); (* Execute this if test = 0; i.e., false case *)
         Bra (BraLabel selJoin);
         Label selTrue;
         Ldd (Index(2,X)); (* Execute this if test != 0; i.e., true case *)
         Label selJoin; (* Join point for two branches *)
         Std (Index(4,X)); (* Store chosen val into former test slot *)
         Inx; (* and pop consequent *)
         Inx;]
| Get ->
   let nonNegLabel = newLabel "get-non-neg"
    in gen
        [Jsr (Ext (AddrLabel "need1")); (* Ensure we at least have an index;
                                          leaves PFSP pointing at index *)
         Ldd (Index(0,X)); (* Load index into D *)
         Bge (BraLabel nonNegLabel); (* Test for negative index *)
         Jmp (Ext (AddrLabel "get-neg-index")); (* Report error for negative index *)
         Label nonNegLabel; (* Reach here if index non-negative *)
         Asld; (* Multiply index by 2 *)
         Addd (Imm (ImmWord 2)); (* Add 2 to account for index slot *)
         Stx (Index(0,X)); (* Copy stack ptr to former top of stack (used as a temp) *)
         Addd (Index(0,X)); (* D <- address of desired element *)
         Jsr (Ext (AddrLabel "check-bounds")); (* Check for out of bounds index *)
         Xgdy; (* Get here if index in bounds; put stack ref in Y *)
         Ldd (Index(0,Y)); (* Load stack elt in D *)
         Std (Index(0,X)); (* Put referenced element at top of stack *)
         Dex; (* and push *)
         Dex;]
| Put ->
   let nonNegLabel = newLabel "get-non-neg"
    in gen
       [Jsr (Ext (AddrLabel "need2")); (* Ensure we at least have an index and value;
                                          leaves PFSP pointing at index *)
        Ldd (Index(0,X)); (* Load index into D *)
        Bge (BraLabel nonNegLabel); (* Test for negative index *)
        Jmp (Ext (AddrLabel "put-neg-index")); (* Report error for negative index *)
        Label nonNegLabel; (* Reach here if index non-negative *)
        Asld; (* Multiply index by 2 *)
        Addd (Imm (ImmWord 4)); (* Add 4 to account for index & value slots *)
        Stx (Index(0,X)); (* Copy stack ptr to former top of stack (used as a temp) *)
        Addd (Index(0,X)); (* D <- address of desired element *)
        Jsr (Ext (AddrLabel "check-bounds")); (* Check for out of bounds index *)
        Xgdy; (* Get here if index in bounds; put stack ref in Y *)
        Ldd (Index(2,X)); (* Load putval in D *)
        Std (Index(0,Y)); (* Store putval in referenced slot *)
        Inx; (* and pop putval *)
        Inx;]
```

Figure 10: Compiling PostFix commands, part 2.

```
| Add -> gen
         [Jsr (Ext (AddrLabel "need2"));
          Ldd (Index (2, X));
          Addd (Index (0, X));
          Std (Index (2, X))]
| Sub -> gen
         [Jsr (Ext (AddrLabel "need2"));
          Ldd (Index (2, X));
          Subd (Index (0, X));
          Std (Index (2, X))]

| (Div | Rem) ->
   let label = newLabel "divRem"
    in gen
       ( [Jsr (Ext (AddrLabel "need2"));
          Clr (Ext (AddrLabel "sign"));
          Ldd (Index (2, X)); (* signed numer in D *)
          Jsr (Ext (AddrLabel "negate-word-by-content"));
          Xgdy; (* unsigned numer in Y *)
          Ldd (Index (0, X)); (* unsigned numer in Y, signed denom in D *)
          Cpd (Imm (ImmWord 0)); (* check for div/rem by 0 *)
          Bne (BraLabel label); (* OK; proceed to calculation *)
          Xgdy; (* Not OK; move unsigned numer to D *)
          Jsr (Ext (AddrLabel "negate-word-by-sign")); (* signed numer in D *)
          Jmp (Ext (AddrLabel (if com = Div then "div0-error" else "rem0-error")));
              (* Report Error *)
          Label label; (* Begin calculation *)
          Jsr (Ext (AddrLabel (if com = Div then
                                 "negate-word-by-content"
                               else
                                 "negate-negative-word")));
               (* Ignore sign of Denom for Rem *)
          Pshx; (* Save pfstk pointer! *)
          Xgdx; (* Numer in Y, Denom in X *)
          Xgdy; (* Numer in D, Denom in X *)
          Idiv] (* Quotient in X, Rem in D *)
       @ (if com = Div then [Xgdx] else []) (* Put quotient or remainder in D *)
       @ [Jsr (Ext (AddrLabel "negate-word-by-sign")); (* signed result in D *)
          Pulx; (* Restore pfstck pointer! *)
          Std (Index (2, X))]) (* Push result on stack *)
```

Figure 11: Compiling PostFix commands, part 3.

```
| P.Mul ->
   gen [Jsr (Ext (AddrLabel "need2"));
        Clr (Ext (AddrLabel "sign"));
        (* Negate rand1 if necessary *)
        Ldd (Index(2,X));
        Jsr (Ext (AddrLabel "negate-word-by-content"));
        Std (Index(2,X));
        (* Negate rand2 if necessary *)
        Ldd (Index(0,X));
        Jsr (Ext (AddrLabel "negate-word-by-content"));
        Std (Index(0,X));
        Ldaa (Index(3,X)); (* Low byte of rand1 in A; Low byte of rand2 in B *)
        I.Mul; (* Multiply lower bytes *)
        Xgdy; (* and store result in Y *)
        Ldaa (Index(3,X)); (* Low byte of rand1 in A *)
        Ldab (Index(1,X)); (* Low byte of rand2 in B *)
        Staa (Index(1,X)); (* Replace low byte of rand2 with low byte of rand1 *)
        Stab (Index(3,X)); (* Replace low byte of rand1 with low byte of rand2 *)
        Ldd (Index(0,X)); (* D contains high byte of rand2 & low byte of rand1 *)
        I.Mul; (* Multiply these bytes... *)
        Stab (Index(0,X)); (* ... and store low byte of product in stack slot 0,
                              which is now unused. *)
        Ldd (Index(2,X)); (* D contains high byte of rand1 & low byte of rand2 *)
        I.Mul; (* Multiply these bytes... *)
        Stab (Index(1,X)); (* ... and store low byte of product in stack slot 1,
                              which is now unused. *)
        Xgdy; (* Move first product back to D *)
        Adda (Index(0,X)); (* Add low byte of second product to high byte of result *)
        Adda (Index(1,X)); (* Add low byte of third product to high byte of result *)
                           (* At this point, D contains unsigned result *)
        Jsr (Ext (AddrLabel "negate-word-by-sign")); (* Give D appropriate sign *)
        Std (Index(2,X)); (* Store product in top stack slot *)]
```

Figure 12: Compiling PostFix commands, part 4.

```
| LT -> compRel "lt" (fun a -> Blt a)
    | LE -> compRel "le" (fun a -> Ble a)
    | EQ -> compRel "eq" (fun a -> Beq a)
    | NE -> compRel "ne" (fun a -> Bne a)
    | GE -> compRel "ge" (fun a -> Bge a)
    | GT -> compRel "gt" (fun a -> Bgt a)
  and compRel relName relFun =
    let trueLabel = newLabel (relName ^ "true")
    and joinLabel = newLabel (relName ^ "join")
     in gen
        [Jsr (Ext (AddrLabel "need2"));
         Ldd (Index (2, X));
         Cpd (Index (0, X));
         relFun (BraLabel trueLabel);
         Ldd (Imm (ImmWord 0));
         Bra (BraLabel joinLabel);
         Label trueLabel;
         Ldd (Imm (ImmWord 1));
         Label joinLabel;
         Std (Index (2, X)) (* Push result on stack *)]

  and compInt n =
    if (n < minInt) then
      raise (CompError ("int smaller than "
                        ^ (string_of_int(minInt))
                        ^ ": " ^ (string_of_int(n))))
    else if (n > maxInt) then
      raise (CompError ("int larger than "
                        ^ (string_of_int(maxInt))
                        ^ ": " ^ (string_of_int(n))))
    else
      gen [Ldd (Imm (ImmWord n))]
```

Figure 13: Compiling PostFix commands, part 5.