CS301 Compiler Design and Implementation **Handout # 19**
Prof. Lyn Turbak **October 3, 2003**
Wellesley College ***Revised October 14, 2003***

Problem Set 3
Due: Friday, October 17

Reading:

- Handout #12: Testing (esp. read about the `MakeTester` functor)
- Handout #14: CONDEX and Desugaring
- Handout #15: Introduction to 6811 Programming (Fred Martin)
- Handout #16: 6811 Instruction Set Summary
- Handout #17: 6811 Instruction Set Details
- Handout #18: Programming the Handyboard

Handouts #15 and #16 were distributed in class. Handout #17 can be found outside my office. (It is also available on-line, but the version outside my office takes one quarter of the 114 pages you would need to print it out on your own!)

Teams:

Work in pairs, and choose partners that you have not worked with before. There are eight students working on this assignment (including Jue), so there will be four pairs.

Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 5pm on the due date (Fri, Oct. 17). The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet);
2. your final versions of `CondexToPostFix.ml` and `CondexToPostFixTest.ml` from Problem 1.
3. your final versions of the assembly programs from Problem 2.

Each team should also submit a softcopy submission of the final `ps3` folder. To do this, execute the following commands in Linux in the account of the team member being used to store the code.

```
cd /students/your-account-name/cs301
cp -R ps3 /home/cs301/drop/your-account-name/
```

Problem 1 [50]: A CONDEX to POSTFIX Translator

As discussed in class, it is possible to write a program that translates CONDEX programs to POSTFIX programs. In this problem, your task is to implement a CONDEX to POSTFIX translator by fleshing out the following three functions in the `CondexToPostFix` module in the `ps3` directory:

- `val transPgm : Condex.pgm -> PostFix.pgm`
  Suppose that *p* is a CONDEX program and `transPgm` *p* yields the POSTFIX program *p'*. Then *p'* should have the same meaning as *p*, in the following sense:
  - if *p* returns an integer *i* when run on argument list *args*, then executing *p'* on *args* should return *i* (i.e., at the top of the stack).
  - if *p* signals an error when run on argument list *a*, then executing *p'* on *a* should signal a similar error.

- `val transExp : Condex.exp -> (int -> int) -> PostFix.com list`
  Suppose that:
  - *e* is a CONDEX expression;
  - *amap* is an **argument map** – i.e., a function that maps CONDEX argument reference indices to POSTFIX stack offsets (as discussed in class);
  - `transExp` *e amap* yields the POSTFIX command list *coms*;
  - *args* is an argument list (list of integers);
  - *s* is a stack (i.e. list) of integers in which `List.nth` *s* `(`*amap j*`)` = `List.nth` *args j* for all *j* in `[1..(List.length` *args*`)]`.

  Then:
  - If evaluating *e* on *args* returns the integer *i*, then executing *coms* on *s* should yield the stack *i* `::` *s*.
  - If evaluating *e* on *args* signals an error, then executing *coms* on *s* should signal a similar error.

- `val transExps : Condex.exp list -> (int -> int) -> PostFix.com list`
  Suppose that
  - `[`e_1`;...;`e_n`]` is a list of CONDEX expressions;
  - *amap* is an argument map;
  - `transExps` *es amap* yields the POSTFIX command list *coms*;
  - *args* and *s* are as in the specification of `transExp`.

  Then:
  - If evaluating e_k on *args* returns the integer i_k for all k in $[1..n]$, then executing *coms* on *s* should yield the stack i_k `::` ... `::` i_1 `::` *s*.
  - If evaluating $e_1 \ldots e_k$ in order from left to right signals an error, then executing *coms* on *s* should signal a similar error.

Notes:

- As usual, start this problem set by performing a `cvs update -d`, and perform an update every time you log in to work on this problem set.
- To load the translator, connect to the `ps3` directory in OCaml via `#cd "/students/`*your-account-name*`/ps3"` and then execute `#use "load-condex-to-postfix.ml"`. As usual, carefully examine the output of the `#use` to see errors that may have scrolled by.
- Of course, you may need to implement auxiliary functions in addition to the three required functions in `CondexToPostFix`.
- The descriptions of `transPgm`, `transExp`, and `transExps` above are specifications (i.e. contracts). They do not say how you must implement the functions, just how the functions must behave.
- For your PostFix implementation, you may either use your solution code from PS2 or my solution code from PS2 (now in the `postfix` directory). The `load-condex-to-postfix.ml` file is configured to do the former, but you can easily change it to do the latter.
- It is recommended that you translate `condex` programs in stages:

  1. First, handle integer literals and arithmetic operators;
  2. Next handle relational operators and conditionals;
  3. Next handle logical operators;
  4. Finally handle argument references.

- You will need to test your translator. Put your testing code in the module `CondexToPostFixTest`. This file can be relatively short because:

  – You can use the `MakeTester` functor described in Handout #12 to simplify the construction of a tester. See the updated `IntexInterpTest`, `CondexInterpTest`, and `PostFixInterpTest` (from the ps2 solutions) for sample uses of `MakeTester`.
  – You can use `CondexInterpTest.entries` for most of your test entries.

  One tricky aspect of testing is handling errors, since the error messages given by Condex and PostFix may not be exactly the same. As noted in class, you can address this by (1) changing the PostFix error message to be consistent with the Condex messages or (2) by translating the Condex error messages to PostFix error messages in your tester. The second approach is more elegant, but more work.

- In order to handle certain out-of-bounds argument references in Condex programs, it is necessary to catch such references during the translation process (because they won't be caught when running the translated program). The following exception has been provided in CondexToPostFix for this purpose:

  ```
  exception TransError of string
  ```

  It should be raised when a statically determinable error is found during the translation process. This exception must be properly handled in the testing code.

- Since the translator deals with two languages that share some of the same constructor names (such as `Pgm`, `Add`, `GT`, etc.), it is necessary to distinguish these by using explicit qualification – e.g., `Condex.Add` vs. `PostFix.add`. To avoid the verbosity of these notations, we can introduce abbreviations for the modules via the following declarations:

  ```
  module C = Condex
  module P = PostFix
  ```

  Now we can instead write `C.Add` and `P.Add`.

- There are several tests in `CondexInterpTest` that highlight the differences between `&` and `&&` and between `|` and `||`. Although you do not have to translate `&&` and `||` (because they are syntactic sugar), you *do* have to understand the differences in terms of translating `&` and `|`. In particular, note the following:

  - `(& 2 3)` evaluates to `1` but `(&& 2 3)` evaluates to `3` (since it's equivalent to `(ifnz 2 3 0)`).
  - `(| 0 4)` evaluates to `1` but `(|| 0 4)` evaluates to `4` (since it's equivalent to `(ifnz 0 1 4)`).

Problem 2 [50]: 6811 Programming

In this problem, you will write the following three 6811 assembly programs for the Handyboard:

1. [15] **gcd** Prompt the user for two unsigned 16-bit numbers in the top row of the LCD and display the GCD of these numbers in the bottom row. The result should also be returned in the D register. For calculating the GCD, use the same iterative algorithm as the following OCaml function:

```
let rec gcd a b =
  if b = 0 then
    a
  else
    gcd b (a mod b)
```

2. [15] **follow-light** When the HandyBoard is appropriately attached to a SciBorg robot (see Handout #18), causes SciBorg to follow a light using the two photosensors at the front of the robot. Use the difference between the photosensor readings to determine which motor is on. Stop when STOP button pressed or when either bumper is pressed.

3. [20] **display-binary** Prompt 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. You should use the same recursive algorithm as in the following OCaml function `displayBinary`:

```
let print = StringUtils.unbufferedPrint

let rec displayBinary n =
  if n <= 1 then
    print (string_of_int n)
  else
    (displayBinary (n/2); print (string_of_int (n mod 2)))
```

For example:

```
displayBinary 0;;
0- : unit = ()
# displayBinary 1;;
1- : unit = ()
# displayBinary 5;;
101- : unit = ()
# displayBinary 6;;
110- : unit = ()
# displayBinary 28;;
11100- : unit = ()
# displayBinary 29;;
11101- : unit = ()
# displayBinary 30;;
11110- : unit = ()
# displayBinary 31;;
11111- : unit = ()
# displayBinary 32;;
100000- : unit = ()
```

Notes:

- Before you begin, make sure you understand all the code in the final version of `hc11/test.asm`, which will be covered in lecture on October 15.
- HandyBoards, Serial Interface/Charger Boards, and SciBorgs can all be found on some bookshelves in SCI E125 (the hardware lab). You should not need to move the transformers (wall warts); some of these are already in the micro-focus. When you are finished with the hardware, please return it to SCI E125 and **make sure you leave the HandyBoard charging** (i.e., the yellow light on the SIBC should be on).
- Write your answers in the file `ps3.asm` within the `ps3` directory. This file already contains a skeleton that loads the library functions in `prolog.asm` and has a frob-knob menu for the three subroutines.
- To download `ps3.asm` onto the HandyBoard:

  1. Make sure you are connected to the `ps3` directory within the OCaml interpreter.
  2. Execute `#use "load-hb.ml"`. (You only need to perform this step once, not every time you do a download.)
  3. Connect the telephone cord from the Serial Interface/Battery Charger to the HandyBoard. The telephone cord connections can be flakey on both ends; you have a connection when the yellow power light goes on on the SIBC.
  4. Put the HandyBoard in download mode by pressing the STOP button when you turn the power switch on.
  5. Execute `Assembler.pad "ps3.asm"` in OCaml. When this function returns, turn the HandyBoard power switch off and then on again.

- Debugging assembly code is challenging! The display routines in `prolog.asm` are very helpful for debugging.
- In `follow-light`, keep in mind that their may be electronic and mechanical bugs in addition to code bugs. Check your analog sensors and switches using the `analog-display` and `digital-display` routines in `test.asm`.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS301 Problem Set 3
Due Friday, October 17

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with on the problem set*):

In the **Time** *column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the* **Score** *column when grading you problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [50]		
Problem 2 [50]		
Total		