CS301 Compiler Design and Implementation — **Handout # 22**
Prof. Lyn Turbak — **November 12, 2003**
Wellesley College — ***Revised November 16, 2003***

Problem Set 5

Due: Wednesday, November 19

Reading:
Carefully study the code for the Bindex and Loopster compilers discussed in class.

Teams:
Work in pairs, and try to choose partners that you have not worked with before.

Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by midnight on the due date (Wed, Nov. 19). The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet);
2. your final versions of `Robster.ml` and `RobsterToHB16.ml` for Problems 1, 2 and 3.
3. your final version of `RobsterToPostFix` (if you choose the Loopster to PostFix translator for Problem 4) or your final version of `PostFixToHB16Typed` (if you choose the dynamically typed PostFix compiler for Problem 4).

Each team should also submit a softcopy submission of the final `ps5` 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 ps5 /home/cs301/drop/your-account-name/
```

Problem 1 [25]: Robster = Loopster + **Robotic Primitives**
In this problem, you will create a robot control language named Robster by extending Loopster with the robotic primitives in Fig. **??**. These primitives are similar to those that are found in "real" robot control languages like HandyLogo and Interactive C. Combining these primitives with the naming, state, and control features of Loopster yields a language in which many simple robot programs are much easier to express than in low-level 6811 assembly code
Below are some sample Robster programs that illustrate the primitives in action.

```
(robster () {SciBorg ping-pong program}
  (seq* (talk-to 3) {Motors 0 and 1}
        (this-way) (on) {Go forward}
        (while true {Infinite loop}
          (seq*
            (while (switch 7) (skip))
            (while (! (switch 7)) (skip))
            (that-way) {Go backward if front bumper pressed}
            (while (switch 8) (skip))
            (while (! (switch 8)) (skip))
            (this-way) {Go forward if back bumper pressed}
            ))))
```

```
(robster () {SciBorg light-following program}
  (seq* (talk-to 3) (this-way) (on) {Go forward}
        (while true
          (if (> (sensor 2) (sensor 3)) {Right eye sees more light}
              (seq* (talk-to 1) (on) (talk-to 2) (off) {Turn right}
                    (clear) (print-string "right"))
              (seq* (talk-to 1) (off) (talk-to 2) (on) {Turn left}
                    (clear) (print-string "left"))
              ))))

(robster (black) {SciBorg line-following program; black is threshold }
  (seq* (talk-to 3) (this-way) (on) {Go forward}
        (while true
          (seq*
            (if (> (sensor 0) black) {Left eye sees black}
                (seq* (talk-to 1) (on) (talk-to 2) (off))) {Turn right}
            (if (> (sensor 1) black) {Right eye sees black}
                (seq* (talk-to 1) (off) (talk-to 2) (on))))))) {Turn left}

(robster () {SciBorg "escape" program}
  (seq* (talk-to 3) (on) (this-way) {Go forward}
        (while true
          (seq*
            (while (switch 7) (skip)) {Wait until front bumper hits obstacle}
            (while (! (switch 7) (skip))) {Wait until front bumper hits obstacle}
            (rd) (wait 20) {Back up for 2 seconds}
            (talk-to 1) (rd) (wait 10) {Turn right for 1 second}
            (talk-to 2) (rd) {Go forward again}
            (talk-to 3)))))
```

A skeleton for the ROBSTER implementation can be found in the `~/cs301/ps5` directory:

- The module `Robster` is a version of the `Loopster` syntax module that has been extended to support the robotic primitives.
- `RobsterInterp` is a version of the `LoopsterInterp` interpreter module that ignores the void primitives and always returns `false` for `switch` and `0` for `sensor`.
- `RobsterTest` includes various ROBSTER test programs, including the ones shown above.
- The module `RobsterToHB16` is a version of the `LoopsterToHB16` compiler that contains skeleton clauses for compiling the `Robster` primitive operators in `compValop` and `compVoidop`.

To complete this problem, flesh out the skeletons of the robotic primitives in `compValop` and `compVoidop`.

Notes:

- You can load ROBSTER into OCAML for the first time via:

```
(#cd "/students/username/cs301/ps5")
(#use "load-robster-all.ml")
```

Valops

S-Expression syntax	OCaml syntax	Description
`(sensor` *sensor-port*`)`	`Sensor`	Returns an integer between 0 and 255 that indicates the value reported by the sensor at port *sensor-port*. There are 8 sensor ports: the ports labelled 0–6 on the lower edge of the HandyBoard, as well as 7 (the frob knob).
`(switch` *switch-port*`)`	`Switch`	Returns `true` if the switch at port *switch-port* is currently "on" and `false` if the switch is "off". There are 18 switch ports: the ports labelled 0–15 on the lower edge of the HandyBoard, as well as 16 (STOP button) and 17 (START button).

Voidops

S-Expression syntax	OCaml syntax	Description
`(wait` *tenths*`)`	`Wait`	Waits (holds control) for *tenths* tenths of a second.
`(clear)`	`Clear`	Clears the LCD display.
`(top)`	`Top`	Set the LCD cursor to the top left.
`(bottom)`	`Bottom`	Set the LCD cursor to the bottom left.
`(talk-to` *motors*`)`	`TalkTo`	Specifies the *current motor set* by a single integer *motors* that is the sum of the *motor tags* of all the motors in the set:

Motor number	Motor tag
Motor 0	1
Motor 1	2
Motor 2	4
Motor 3	8

		E.g. `(talk-to 13)` established Motors 0, 2, and 3 as the current motor set because $13 = 1 + 4 + 8$. When the HandyBoard is turned on, the current motor set is empty.
`(on)`	`On`	Turns on all motors in the current motor set.
`(off)`	`Off`	Turns off all motors in the current motor set.
`(toggle)`	`Toggle`	Toggles the on/off state of all motors in the current motor set – i.e., it turns off motors that are currently on and turns on motors that are currently off.
`(this-way)`	`This`	Sets the directions of all motors in the current set to "this way" (i.e., represented by the green LED). The LED will only be lit if the motor is also on.
`(that-way)`	`That`	Sets the directions of all motors in the current set to "that way" (i.e., represented by the red LED). The LED will only be lit if the motor is also on.
`(rd)`	`Rd`	Reverses the direction of all motors in the current motor set.

Figure 1: Robster robotic primitives.

This will load many files. After executing the above once, you can reload just the four Robster modules `Robster`, `RobsterInterp`, `RobsterToHB16`, and `RobsterTest` by executing `(#use "load-robster.ml")`.

- To parse, compile, assemble, and download the sample programs in `RobsterTest`, execute `RobsterTest.ctest RobsterTest.`*name*, where *name* is the name of the program string in `RobsterTest`. In addition to the sample programs shown above, `RobsterTest` also includes the following test programs that are very helpful for testing your robotic primitives: `analogTest`, `digitalTest`, `onOffTest`, `toggleTest`, `thisThatTest`, and `rdTest`.
- Many primops can be implemented by just invoking an appropriate subroutine in the library file `~/cs301/hc11/prolog.asm`. For other primops, it is helpful to define new subroutines in your compiled code.
- Many of the problems involve manipulating particular bits in a byte. For this manipulations, you will find the following operations helpful: `anda/andb`, `oraa/orab` (inclusive or), and eora/eorb (exclusive or).
- For the `switch` primitive, note the following:
  - Analog sensor ports 0–6 can be interpreted as switches by treating values of 0–127 as true and 128–255 as false.
  - Digital switch ports 7, 8, and 9 come from bits 1, 2, and 8 (respectively) of the byte read from `porta` (address `$1000`).
  - Digital switch ports 10–15, STOP (switch 16), and START (switch 17) come from bits 1–8 (respecively) of the byte read from `digital-in` (address `$7000`).
  - You must report an error if the argument to `switch` is out of the range 0–17.
  - Try to avoid using too many tests and branches in your implementation of `switch`.
  - When testing your `switch` primop with `test-digital`, testing the START button is a bit tricky because of the way `wait-for-start-stop` is used to handle printing operations in the rest of the compiler.
- For the `motor` primitive, note the following:
  - Motors are controlled by writing a byte to `motor-port` (address `$7000`). The lower four bits of the byte control direction of the four motors (0 = this-way, 1 = that-way), while the upper four bits of the byte control the on/off status of the four motors (0 = off, 1 = on).
  - The motor commands `on`, `off`, `toggle`, `this-way`, `that-way`, and `rd` only affect the motors in the motor set established by the most recent call to `talk-to`. The state of any motor not in this set is not affected by any of these commands.
  - You must report an error if the argument to `talk-to` is out of the range 0–15.

Problem 2 [20]: Exit and continue

In programming languages with looping constructs, it is sometimes helpful to alter the normal flow of control through a loop. In this problem, you will extend ROBSTER with the following two loop control statements:

S-Expression syntax	OCaml syntax	Description
`(exit)`	`Exit`	Exits from the innermost enclosing `while` loop without executing any more of the loop body or test code.
`(continue)`	`Cont`	Jumps to the "top" of the innermost enclosing `while` loop (i.e., the loop test) without executing the rest of the loop body in the current iteration.

It is an error if `exit` or `continue` are invoked in code that is not inside the body of a `while` loop. This error can be detected at compile time.

As an example of `exit` and `continue` in action, consider the following ROBSTER program:

```
(robster (lo hi)
  (decl sum 0
    (decl i lo
      (seq
        (while (<= i hi)
          (seq*
            (if (= (% i 42) 0) (exit)) {Stop at multiples of 42}
            (if (= (% i 13) 0) {Exclude multiples of 13}
                (seq (<- i (+ i 1)) (continue)))
            (<- sum (+ sum i))
            (<- i (+ i 1))))
        (print-result sum))))
```

The program sums the numbers between `lo` and `hi`. Multiples of 13 are excluded from the sum by using `continue` to avoid updating the sum in this case. Whenever a multiple of 42 is encountered, the program exits immediately with the current sum. For example:

lo	hi	result
1	10	55 (= (1+10)*(10/2)
10	30	381 (= (10+30)*(21/2) - 13 - 26)
38	50	119 (= 38 + 40 + 41)
80	100	326 (= 80 + 81 + 82 + 83)

Note that `exit` and `continue` only affect control in the innermost enclosing `while` loop and have no effect on any other enclosing `while` loop. For example, consider the ROBSTER program with nested `while` loops in Fig. ??. When invoked on the input `5`, this program prints out the following sequence of integers:

```
11 21 31 33 41 43 51 53
```

In this problem, you are to implement `exit` and `continue` within the `RobsterToHB16` compiler by modifying `compStm` to (1) pass any additional parameters necessary to handle the new constructs and (2) flesh out clauses for `Exit` and `Cont`. Your compiler should raise a compile time error (using `CompError`) if `exit` or `continue` are ever encountered outside a `while` loop body.

```
(robster (n)
  (decl i 1
    (while (<= i n)
      (decl j 1
        (seq
          (while (<= j i)
            (seq* (if (= (% j 4) 0) (exit))
                  (if (= (% j 2) 0)
                      (seq (<- j (+ j 1)) (continue)))
                  (clear)
                  (println-int (+ (* 10 i) j))
                  (while (switch 16) (skip))
                  (while (! (switch 16)) (skip))
                  (<- j (+ j 1))))
          (<- i (+ i 1)))))))
```

Figure 2: An example with `exit` and `continue` within nested `while` loops

Problem 3 [15]: Desugaring

Many Robster programs can be simplified by introducing some new syntactic sugar.

Sugar	Description
`(loop` S_{body}`)`	Executes the body statement S_{body} in an infinite loop. Within the `loop` body, `exit` exits the loop and `continue` starts the next iteration of the loop.
`(wait-until` E`)`	Waits for the expression E to change value from false to true. If E is initially true, `wait-until` will not return until E first becomes false and then true again. This is known as *edge-triggered logic.*
`(++` I`)`	Increments the mutable integer variable I.
`(+=` I E`)`	Modifies the mutable integer variable I by adding to it the value of the integer expression E.
`(for` I E_{lo} E_{hi} S_{body}`)`	Executes the body statement S_{body} for each of the values of the integer index variable I between E_{lo} and E_{hi}. The integer expressions E_{lo} and E_{hi} should be evaluated exactly once, before S_{body} is executed for the first time. Within the S_{body} of a `for` loop, `exit` exits the `for` loop and `continue` starts the next iteration of the loop (in which case it is necessary to first explicitly increment the index variable).

Fig. 2 presents some examples of Robster programs from above that have been simplified by using the new syntactic sugar.

In this problem, you are to extend the desugaring rules `desugarRules` of Robster in the module `Robster` to handle the five new syntactic sugar forms introduced above. You can test your desugaring by executing `RobsterTest.dtest RobsterTest.`*name*, where *name* is the name of the program string in `RobsterTest`.

```
(robster () {SciBorg ping-pong program}
  (seq*
    (talk-to 3) {Motors 0 and 1}
    (this-way) (on) {Go forward}
    (loop {Infinite loop}
      (seq*
        (wait-until (switch 7))
        (that-way) {Go backward if front bumper pressed}
        (wait-until (switch 8))
        (this-way) {Go forward if back bumper pressed}
        ))))

(robster (lo hi)
  (decl sum 0
    (for i lo hi
      (seq*
        (if (= (% i 42) 0) (exit)) {Stop at multiples of 42}
        (if (= (% i 13) 0) {Exclude multiples of 13}
            (seq (++ i) (continue)))
        (+= sum i)))
    (print-result sum)))

(robster (n)
  (for i 1 n
    (for j 1 i
      (seq* (if (= (% j 2) 0)
                (seq (++ j) (continue)))
            (if (= (% j 4) 0) (exit))
            (clear)
            (println-int (+ (* 10 i) j))
            (wait-until (switch 16))))))
```

Figure 3: Robster programs using the new syntactic sugar.

Problem 4 [40]: Your Choice

In this part, you have a choice between two problems. You only need to do one of the two problems, although you are welcome to do the other one for extra credit.

4a: Dynamic Typing of Compiled PostFix

In class on Wed. Nov. 12, we discussed how to modify the `PostFixToHB16` compiler from PS4 so that the compiled code dynamically catches PostFix type errors when it is executed. The key idea is to use the least significant bit of a 16-bit word as a "type tag", where 0 is the tag for an integer and 1 is the tag for a pointer. Because of the tag bit, only 15-bit signed integers can be represented in this scheme.

The file `~/ps5/PostFixToHB16Typed.ml` initially contains Lyn's solution to PS4 (the PostFix to HandyBoard compiler that does *not* perform dynamic type checking). Modify this file so that it implements a PostFix to HandyBoard compiler that *does* perform dynamic type checking. Follow these guidelines:

- The 6811 parser and assembler have been extended to handle a new alignment declaration that you will need for this problem. In "raw" 6811 assembly the directive is written:

  `align` *r d*

  while in OCaml abstract instruction syntax the directive is written

  `Align (`*r*`, `*d*`)`

  In either case, the alignment directive means to insert zero or more `nop` instruction bytes (opcode = 1) so that the address of the first byte after the alignment directive is at an address *a* that gives a remainder *r* when divided by *d*.

  For example, the following table shows the number of `nop` bytes inserted by alignment calls at various addresses:

Address	`align 0 2`	`align 1 2`	`align 0 4`	`align 1 4`	`align 2 4`	`align 3 4`
`$9000`	0	1	0	1	2	3
`$9001`	1	0	3	0	1	2
`$9002`	0	1	2	3	0	1
`$9003`	1	0	1	2	3	0

- When the translated code encounters a dynamic type error, it should display an error message on the LCD of the HandyBoard and then restart the program.

- Test your translator on a wide variety of PostFix programs to make sure it behaves appropriately. You should check that it works in non-error cases as well as error cases. In particular, test that the 15-bit signed arithmetic works as expected.

- Execute `(#use "postfix-to-hb16-typed.ml")` to load all files required for this problem and `(#use "PostFixToHB16Typed.ml")` to load only the translator file.

4b: A Robster to PostFix translator.

In PS3, you implemented a Condex to PostFix translator. In this problem, you will implement a Robster to PostFix translator "from scratch" in the file `RobsterToPostFix.ml`. Executing (in a PostFix interpreter) the PostFix program that results from translating a Robster program should have the same effect as executing the original robster program in a Robster interpreter.

Follow these guidelines:

- The most interesting aspects of the translation are handling variables and loops. Think carefully about how you will handle these before you begin your implementation. In particular, you will need some sort of static environment that models the current offset of every temporary value and variable on the PostFix stack.
- A Robster program does not return any value but is executed for its effect. But a PostFix program is required to end with an integer value at the top of the stack. A translated Robster program that executes without error should end with the integer `0` at the top of the stack.
- Because of limitations in PostFix there are several Robster operations that cannot be faithfully implemented by the translator. You should handle these as described below:
  - PostFix cannot handle any of the robotic primitives from Problem 1. Your compiler should treat any robotic void ops as `skip`s and should treat `sensor` as if it always returns `0` and `switch` as if it always returns `false`.
  - You do not have to handle the `exit` and `continue` constructs from Problem 2. Treat each of these as if it is a `skip`.
  - As currently configured, PostFix cannot handle `string-length` or `string-get`. You can treat `string-length` as if it always returns `0` and `string-get` as if it always returns `'a'` (ASCII 97).
  - It is possible for PostFix to handle to `print-char`, but you are not require to do so in this problem. (You may do so for extra credit.)
- You may assume that the Robster program being translated is "type-correct" – i.e., you do not have to worry about any type errors in the Robster program.
- As of this writing, there is no automatic way to compare the results printed by executing the PostFix program that results from translating a given Robster program with the results printed by executing the original Robster program.
- Execute `(#use "robster-to-postfix.ml")` to load all files required for this problem and `(#use "RobsterToPostFix.ml")` to load only your translator file.

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

CS301 Problem Set 5
Due Wednesday, November 19

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 [25]		
Problem 2 [20]		
Problem 3 [15]		
Problem 4 [40]		
Total		