CS301 Compiler Design and Implementation **Handout # 21**
Prof. Lyn Turbak **October 24, 2003**
Wellesley College

Problem Set 4

Due: Tuesday, November 4

This is a draft of PS4 that contains the description of the first of the two problems.

Reading:

Carefully study the code for the compilers discussed in class.

Teams:

Work in pairs, and choose partners that you have not worked with before. There are eight students working on this assignment, so there should 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 (Tue, Nov. 4). The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet);
2. your final version of `PostFixToHB16.ml` from Problem 1.
3. your final versions of `LoopsterToPostFix.ml` and `LoopsterToPostFixTest.ml` from Problem 2.

Each team should also submit a softcopy submission of the final `ps4` 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 ps4 /home/cs301/drop/your-account-name/
```

Problem 1 [60]: A PostFix to 6811 Translator

In this problem, you will implement a compiler that translates from PostFix to 6811 assembly code that runs on the HandyBoard. Your task is to flesh out the following function in the `PostFixToHB16` module in the `ps4` directory:

> `val compile : PostFix.pgm -> Instr68HC11.instr list`
> Suppose that p is a PostFix program with n arguments and `compile` p yields the OCaml representation of an 6811 instruction list. When these instructions are executed on the HandyBoard, the user should be prompted for n argument words (not bytes). All values manipulated by the program should be 16-bit words. If p executes without error, then the integer result at the top of the final stack should be displayed on the bottom of the LCD display. Handling a program with an error is more complex, and is discussed in more detail below.

Additionally, the following helper functions have already been provided in `PostFixToHB16`:

- `val cad : PostFix.pgm -> unit`
  Compile, assemble, and download the given PostFix program.
- `val pcad : string -> unit`
  Parse, compile, assemble, and download the given string representation of a PostFix program.
- `val pcu : string -> unit`
  Parse, compile, and unparse a string representation of PostFix program, displaying an assembly code listing of generate 6811 assembly code. Examining the assembly code your compiler produces is an important debugging tool.

Conventions:

As usual when writing any compiler, it is important to establish the conventions that your compiler will observe. In a PostFix to 681111 compiler, the most important decision is how to represent the PostFix stack. It turns out that you *cannot* use the standard system control stack for the PostFix stack, because the control stack is needed to handle the execution of PostFix executatble sequences in a way that is incompatible with the PostFix stack. So the PostFix stack must go elsewhere in memory.

It is recommended that you place the PostFix stack so that the bottom of the PostFix stack is at address `$bf00` (before the reset vector) and that the stack grows "down" in memory, toward your compiled code. As noted in the *Errors* section below, you should guarantee (1) that the stack never grows so big that it overwrites your compiled code and (2) that the stack never goes above `$bf00`, where it might overwrite the reset vector.

You will need to keep track of the PostFix stack pointer (PSP), the address of the top element of the PostFix stack. It is recommended the X register be reserved for storing the PSP. This will allow you to used indexed operations involving the X register to access elements on the stack. Recall that the offsets in indexed operations are always positive, so indexed operations involving the PSP can refer only to memory addresses at the PSP and above. This is why it is crucuial that the stack grow downwards to lower memory addresses rather than growing upwards; otherwise it would not be possible to use indexed operations to access elements on the PostFix stack.

You have a choice of whether the PSP should point to the top value on the stack or to the next free slot at the top the stack. It does not matter which you choose as long as you are consistent.

When manipulating the stack, keep in mind that all quantities stored on the stack will be one word = two bytes, which takes two memory locations. The 6811 convention is to store the high byte of the word at the lower of the two addresses.

In addition to handling integers, the PostFix stack also needs to handle strings and executable sequences. Each of these can be represented as a 16-bit quantity:

- A string can be represented by the address in memory where the string is stored.
- An executable seqeunce can be represented by the address in memory of a subroutine that, when invoked, will execute the code of the executable sequence.

You have to decide where you will store your strings and executable sequence subroutines. One approach is to store them "in-line", in the middle of the compiled code, and have the compiled code jump around them. Another approach is to store them after (or before) the compiled code so that no jumps are required to avoid them.

Errors:

Recall that there are many kinds of errors that can be encountered during the execution of a PostFix program. We can broadly classify these into two categories:

1. **Stack Errors**: These are errors in which the stack does not have the right size to perform the operation. For example:
   - `pop` on an empty stack;
   - `swap` or `add` or `gt` on a stack with less than two elements;
   - `sel` on a stack with less than three elements;
   - `get` or `put` with an index that is out of bounds on the stack.
2. **Type Errors**: These are errors in which the stack has enough elements, but they are the wrong type. For example:
   - one of the operands of `add` is not a number (i.e., it's a string or executable sequence);
   - the discriminant of a `sel` is not a number;
   - the index of a `get` or `put` is not a number;
   - the operand of `prs` is not a string;
   - the operand of `exec` is not an executable sequence.

For reasons to be discussed in class, it is difficult to catch the type errors without making major representational changes. To simplify the compiler, you may assume that your compiled code does *not* need to catch and report any type errors.

However, your compiled code *must* catch and report any stack errors (e.g. by displaying an error message on the LCD display). The simplest approach is to display a "stack too small" message in these cases, though a message indicating the offending operation would be more helpful. In addition to the usual errors involving stacks that do not contain enough elements, you must also handle the case in which the stack grows too big. That is, you cannot allow stack pushes to overwrite any of your compiled code, and should report a "full stack" if any attempt is made to do this.

Notes:

- To load the compiler, connect to the `ps4` directory in OCaml via `#cd "/students/`*your-account-name*`/ps4"` and then execute `#use "load-postfix-to-hb16.ml"`. You should only do this *once* for every interactive programming session. Thereafter, you should just execute `#use "PostFixToHB16.ml"` every time you change the compiler.

- In class, values manipulated by the compiler were 8-bit bytes, but on this assignment the values manipulated by the compiler are 16-bit words (where numbers are signed). This affects the instructions you will use for arithmetic calculations and stack manipulation. Multiplication is particularly tricky, since you want to multiply two 16-bit signed numbers, but the 6811 `mul` instruction multiplies two 8-bit unsigned numbers. Think about how multiplication works and develop a strategy for using several `mul` operations to implement multiplication of two 16-bit signed numbers.

- Since some PostFix command names conflict with some 6811 instruction names, you will sometimes have to use qualified constructors. The following module abbreviations have been provided for you in `PostFixToHB16`:

```
module P = PostFix
module I = Instr68HC11
```

- As of this writing, you should manually test the compiled code on the HandyBoard. Lyn will try to develop a more automated testing process.

Problem 2 [40]: A Loopster to PostFix Translator

In this problem you will implement a translator from the Loopster language to PostFix. Together with your compiler from Problem 1, this gives you a way to compile Loopster programs to HandyBoard programs that use 16-bit signed words. (The Loopster compiler that will be described in class will use 8-bit signed words.)

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

CS301 Problem Set 4
Due Tuesday, November 4

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 [60]		
Problem 2 [40]		
Total		