CIS 240 Fall 2014

Homework #9: Stack Attack {200 pts}

Due dates:
Milestone assignment: Midnight Wednesday Dec. 3
Final assignment: Midnight Tuesday Dec. 9

Preamble:

In this assignment you are going to write a small compiler that will transform code written in a new language, called J, into LC4 assembly code much the way that the lcc compiler converter converts C code into assembly. **Your compiler will be designed to use exactly the same calling convention that lcc does that way your J programs can call subroutines compiled by lcc and vice versa**.

Once again you are expected to do all of your work on eniac using the clang compiler. That is where your code will be tested and graded. You are expected to provide all of the C source code and a makefile for producing the final executable program, jc. This program will act much like lcc does, when provided with a properly formatted source file in the J language it will produce the corresponding assembly code. That is

```
>>  jc foo.j
```

will produce a new file called foo.asm if foo.j contains an acceptable program otherwise it will print out helpful error messages.

The J Language

The J Language is a functional language with a syntax inspired by, but not identical to, classical s-expression based languages like LISP and Scheme. The compilation process can be conceptually divided into 3 stages: a tokenization stage where the input ASCII file is converted into a stream of tokens that are meaningful in the language, a parsing stage which enforces the syntax and structure of the language, and a code generation process which generates the requisite assembly instructions as the program is being parsed.

For this assignment we are providing you with code to do the tokenization and parsing. Your job is to understand how the code we gave you works and use it to implement the jc compiler.

Like C the J language is procedural in the sense that all of the code is packaged in functions. The example below shows a simple J function:

```
;; This is a comment
(defun foo (a b)
	(let c (+ a b))
	(if (> c 10) (* 2 a) (* b b)))
```

As shown in this example every function definition is enclosed in parentheses and begins with the **defun** keyword followed by the function name and a parameter list which can be empty if there are no parameters. The function body consists of a series of expressions and the return value of the function is simply the value of the last expression. Your functions should be implemented using the same calling convention as lcc, you should refer to the lecture files to learn about the way that parameters are passed in to a function, the way that return values are passed out and the way that the stack is handled to allow functions to call into and return from each other successfully.

To simplify matters the only data type in the J language is the 16 bit word, the native data type of most LC4 instructions.

J Expressions:

The table below lists the various kinds of expressions that you can encounter in J. Every expression produces a single 16 bit value which it leaves at the top of the stack. As we mentioned previously, the value returned from a function is simply the value of the last expression.

NUMBERS: The J language allows for both signed decimal integers egs 234, -780 and unsigned hexadecimal values egs 0xAB. The value of this kind of expression is just the numerical value.

IDENTIFIERS: egs my_value, foo_fighter, ThisIsAValue. The J language allows you to define identifiers and associate values with them via the let statement. When an identifier is encountered in the code the value associated with it should be placed atop the stack. In J identifiers can consist of letters, digits and underscore characters but they must start with a letter.

LET EXPRESSIONS: egs (let my_value (+ 3 4)) Let expressions associate values with identifiers. The second element in the s-expression must be an identifier and the third element is an expression whose value will be bound to the identifier in the remainder of the function. The value returned by a let expression is the value that is ultimately bound to the identifier. **Note that let expressions only bind identifiers within the scope of the enclosing function** so they play the same role as local variables in a C function. The only difference is that since J is a functional language, these identifiers can only be bound once in a function.

BUILT IN OPERATORS: The J language provides a range of built in binary operations which are expressed in prefix notation. Egs (+ 5 6), (% 7 4). These are listed below along with the value that they should produce.

(+ expr1 expr2)	Return (expr1 + expr2)
(- expr1 expr2)	Return (expr1 – expr2)
(* expr1 expr2)	Return (expr1 * expr2)
(/ expr1 expr2)	Return (expr1 / expr2)
(% expr1 expr2)	Return (expr1 % expr2)
(> expr1 expr2)	Return (expr1 > expr2)
(< expr1 expr2)	Return (expr1 < expr2)
(>= expr1 expr2)	Return (expr1 >= expr2)
(<= expr1 expr2)	Return (expr1 <= expr2)
(== expr1 expr2)	Return (expr1 == expr2)

In every case the operation should expect its two operands to be waiting for it on the top of the stack. It should pop these two values, perform the requisite operation and leave its result at the top of the stack. This strategy will allow us to nest expressions recursively egs. (+ (* a b) (- 6 (/ 7 f)))).

For simplicity the '/' and '%' operations should treat their operands as unsigned values just the way the corresponding LC4 instructions do.

IF EXPRESSIONS: In J the if statement is the only control structure that we need. It is an s-expression where the first element is the keyword "if" the second element is an expression which is evaluated to decide what to do next, if the result is non-zero the third expression is evaluated and it's result is returned as the expression value otherwise the fourth element is evaluated and returned. Note that this means that every if statement must effectively have an 'else' clause.

Egs (if (> a b) a b)) returns the greater of a and b.

FUNCTION INVOCATIONS: In J you invoke a function in an s-expression where the first element is an identifier giving the name of the function and the remaining expressions are evaluated and passed as parameters. Here is an example:

(my_boss_function param1 (+ 3 4))

Note that you must use the same calling convention as lcc so parameters must be placed on the stack in the order that lccc would. The value returned by the function invocation should be placed on the top of the stack. Think carefully about what the stack looks like when a C function returns to its caller and reason about how it would need to be updated to achieve the desired result.

NOTE: The J language is case sensitive so you need to ensure that your code respects the case expected when using identifiers and keywords.

Recursive Descent Parser:

We have provided you with a recursive descent parsing library which can be used to parse J programs. The code is contained in the files parser.h and parser.c, this code depends on the tokenization routines provided in token.h and token.c. We also provided the test programs test_parser.c and test_token.c so that you could see how this code works. You need to study this code carefully to understand how it works. You can find more information on recursive descent parsers here:

http://en.wikipedia.org/wiki/Recursive_descent_parser

Note that parser.c contains functions corresponding to the major non-terminal features of the language and you will need to add additional code to these routines to emit the requisite assembly program.

You will note that parser.h defines an enumerated type which is a simple and elegant way to handle situations where you want to use variables that can only take on a finite range of values.

Keeping track of named values

Note that the J language has many of the same features as the C language. It's a procedural language with named functions. The functions accept parameters and return values. Through the let mechanism you can create named values in your program. In order to support this you will need to use a data structure like a list or a heap to keep track of the identifiers that you encounter while you are parsing a function. This will allow you to record important information such as where you stored those values on the stack. You should also use this data structure to detect syntactic errors like using the same identifier twice in two let statements or using a parameter name in a let statement or using an identifier in an expression before it has been bound in a let statement.

EBNF description:

Extended Backus-Naur Form or EBNF is a simple format for succinctly describing the format of a language. You can find more information on EBNF at the link above. An EBNF description of the J language is provided below:

```
ident = letter { letter | digit | “_”};

decimal_value = [-] digit {digit};

hex_value = “0x” hexdigit {hexdigit};

expr = decimal_value | hex_value |  ident | sexpr;

sexpr =
“(“ “+”  expr expr “)” |
“(“ “-” expr expr “)” |
“(“ “*” expr expr “)” |
“(“ “/” expr expr“)” |
“(“ “%” expr expr “)” |
“(“ “>” expr expr “)” |
“(“ “<” expr expr “)” |
“(“ “<=” expr expr “)” |
“(“ “>=” expr expr “)” |
“(“ “==” expr expr “)” |
if_statement | function_call;

let_statement = “(“ “let” ident expr “)”;

if_statement = “(“ “if” expr expr expr “)”;

function_call = “(“ ident  {expr} “)”;

function = “(“ “defun”  ident “(“ {ident} “)”
{let_statement}
expr
{expr}
“)”;

program = {function};
```

Comments in J:

Note that when a semicolon character ‘:’ is encountered everything between that location and the end of the line is treated as a comment.

Milestone Assignment: {10 points}

The milestone assignment is a pencil and paper task that is intended to get you started thinking about the problem. The assignment consists of 2 parts. First write the Euclidean Algorithm for computing the GCD of two integers using the J language. Then generate, by hand, the assembly program that your jc program would produce given the J function that you wrote. You will turn your answers in to a text box on Canvas.

NOTE: Because this is a milestone assignment it cannot be turned in late. If it is not submitted by the stated due date you get 0 points for this component of the assignment.

Testing:

As usual you can't really tell if your code works if you don't test it. We will be providing examples of J and C code that you should be able to compile and link together to form working executables that run in PennSim. We will also be providing opportunities to write more sophisticated C and J code for extra credit.

You would be well advised to write your own test cases as well to test your code since we will probably be testing on more than just the examples that are handed out.

Submission instructions.
Create a directory containing all of your code files. You should name the directory as follows FIRST_LAST_HW_##. For example CHRIS_BROWN_HW4. If you have a long first or last name feel free to shorten it if you can do so without ambiguity. Create a compressed version of your directory. On windows you can sometimes do this by right clicking on the directory and then selecting "Send To -> Compressed Folder". On the Mac you can right click on the directory and select "Compress". Once you have compressed your directory into a single zip file you can submit it on Blackboard as per usual.

Your directory should contain the following items:

1. All of the source code files required to build the jc program.
2. A Makefile as specified above that can be used to build the jc program
3. A README file containing notes for the TAs if necessary

We will use the Makefile you provide to compile your code. Then we will test the code with our own inputs. If the code doesn't compile or doesn't run you should not expect many points.

Please make sure that you submit the latest versions of your code.