CS61c Summer 2014 Project 1: LIFC Compiler

Due: July 18 at 23:59:59

Our first project is to write a compiler from the homebrew language of LIFC (LIFC Is For Compiling) to MIPS assembly. This compiler will take a source file and produce MAL[1] assembly code, which you can then feed to MARS[2] for testing. Compilers is an interesting[3] topic in its own right, but we've taken measures to ensure that the only things you'll need to complete this project is a little ingenuity and some elbow-grease. Your job is to produce a specification compliant compiler, including adequate internal documentation (comments), and a thorough set of test cases.

1 Setting Up Your Workspace

We'll be keeping the skeletons for this semester's projects at `git@github.com:ucberkeley-cs61c/projects_su14.git`, so you can get a copy with the following commands

```
$ cd ~/work #Assumes you placed your working repository at ~/work
$ git remote rm projects #Only needed if you've already messed up this alias
$ git remote add projects git@github.com:ucberkeley-cs61c/projects_su14.git
$ git pull projects master
```

Once you've pulled the skeleton you should find a folder named `proj1` inside your local repository. This folder contains the skeleton for this project.

2 Running Your Solution

The name of the final executable which we'll be testing when we look at your submission is `lifcc` (short for LIFC Compliler). The executable should be runnable via the command

```
$ ./lifcc PROG.lc
```

which should output a MIPS program to stdout with the same behavior as the LIFC program defined in `PROG.lc`. To place the results into a file simply use bash's `>` functionality. E.g.

```
$ ./lifcc PROG.lc > PROG.s
```

3 The LIFC Language

LIFC is a language which mimics the syntax of lisp for easy parsing and the semantics of C for easy code generation. Like scheme everything in LIFC has a value (even if that value is `None`), but unlike scheme everything in LIFC corresponds 1-1 with a construct in C, and many of the more troublesome features are done away with entirely (e.g. returning functions as values).

[1] MIPS code which may or may not contain pseudoinstructions.

[2] The assembler+simulator we'll be using to run MIPS code in this class. See `http://courses.missouristate.edu/kenvollmar/mars` for details.

[3] hard

3.1 Syntax and Lexing

The foundation of LIFC syntax is the *expression*, a computation which returns a value. A LIFC program takes the form of a sequence of one or more expressions which are evaluated in order. These expressions may either take the form of a *parenthetical* or an *atom*. Parentheticals take the form of `(OPERATOR OP1 OP2 ...)` where the operator must be an atom but operands may be either atoms or parentheticals (which may themselves have parenthetical arguments, and so on and so forth). For instance, `(- (+ 1 2) (* 3 4))` calculates and returns the value of $1 + 2 - 3 * 4$.

Atoms are either string literals, integer literals, variable names, function names, or keywords:

Integers A native type which corresponds to a signed 32-bit 2's complement number. Decimal numerals are the LIFC syntax for integer literals (e.g. `42`, `-42`, but NOT `+42`). Behavior is undefined if a decimal numeral is provided which cannot be represented by a 32-bit 2's complement number.

Strings A native type which corresponds to a null terminated ASCII string. `string`s, like `struct`s, are passed by reference (see the keywords section for more details). The syntax for a string literal in LIFC is `"A STRING"`. LIFC uses the same conventions for escaping characters as most other languages, including C.

Keywords A complete list of keywords, and their meanings, is presented in the following subsection.

Variables A variable must match the regexp `[a-zA-Z_][a-zA-Z0-9_]*`, and must not match a keyword. It can, however, contain a keyword.

Function Name A function name must match the regexp `[a-zA-Z_][a-zA-Z0-9_]*`, and must not match a keyword. It can, however, contain a keyword.

Atoms must be separated either by whitespace or by parentheses.

3.2 Keywords

LIFC defines the following keywords:

and,or Basic logical operations. The syntax of these is `(and OPERAND1 OPERAND2)` and `(or OPERAND1 OPERAND2)`. The only value which evaluates to false is `None`, and the return value should be either 1 or 0, depending on the value computed. These operations should short-circuit – the second operand should be evaluated only in the event that the first operand is insufficient to determine whether or not the result should be true.

+,-,*,/ Basic arithmetic operations. These have syntax of the form `(OP OPERAND1 OPERAND2)`.

lt,eq Basic comparators. These have syntax of the form `(OP OPERAND1 OPERAND2)`, and should always return logical values.

function Allows for the definition of functions inside a LIFC program. Any call to the `function` utility must occur at the topmost level of the program, and a function must be defined via the `function` utility before it is referenced inside of a program. The syntax for declaring a function is as follows: `(function (FUNC_NAME ARG1 ARG2 ...) FBODY)` where `FUNC_NAME` is the name of the function being defined, `ARG#` is the name of the #th argument, and `FBODY` is the expression to evaluate during a call to `FUNC_NAME`. Functions may have as few as 0 arguments. It is illegal to define two functions with the same name, but a function may have the same name as a variable. The return value of a function call is the value of its body. The `function` utility itself should return `None`.

The syntax for calling a function is analogous to the syntax for invoking most keywords and takes the form `(FUNC_NAME VAL1 VAL2 ...)`, where `VAL#` is the #th argument to `FUNC_NAME`.

struct The `struct` keyword allows you to package multiple data together. The syntax for a `struct` statement is `(struct VAL1 VAL2 ...)`, where `VAL#` is the value of the `#`th field. `struct`s are pass-by-pointer in LIFC. That means that if I pass a `struct` to a function, or assign it to a variable then I'm really assigning a pointer.

arrow Extracts a field from a `struct`. The syntax for `arrow` is `(arrow STRUCT FIELD_NUM)`. For example, `(arrow (struct 3 (+ 2 3) "hi") 2)` would return `"hi"`. Yes, this is all kinds of ugly, but it allows us to keep user-defined data types without having to add any typing to the language or keeping a lookup table inside each of our `struct`s.

Behavior is undefined in the event that the first operand to `arrow` is not a struct or if the second operand is not a non-negative integer smaller than the number of fields in the first operand.

None Equivalent to an integer literal of 0. The only value which is logically false.

assign Assigns a new value to a variable. Declares the variable if there is not already a variable by the same name in scope. The syntax for `assign` is `(assign VARIABLE_NAME VALUE)`. Returns the value of the variable following the assignment.

Although we will not be testing for this when grading your project, the assignment operator may also be used to assign into a struct field by pairing it with the arrow operator. E.g.

```
(assign x (struct 1 2))
(assign (arrow x 0) 4)
```

would assign a value of 4 into the second field of the `struct` to whicht `x` points.

if Conditionally evaluates one of two expressions, depending on the value of its predicate. The syntax for an `if` expression is `(if PREDICATE IF_TRUE IF_FALSE)`. An `if` expression should return the value of whichever expression it evaluates.

while Repeatedly evaluates an expression until a stopping criterion is met. The syntax for this is `(while PREDICATE BODY)`. A while-loop should return `None`.

for Repeatedly evaluates an expression until a stopping criterion is met. The syntax for this is `(for INIT PREDICATE INCREMENT BODY)`. A for-loop should return `None`.

sequence Evaluates a sequence of expressions and returns the value of the last one. Has the following syntax: `(sequence EXPR1 EXPR2 ...)`. A `sequence` must contain at least one expression.

intprint Prints its argument as an integer and returns `None`. Has the syntax `(intprint ARG)`.

stringprint Prints its argument as a string and returns `None`. Has the syntax `(stringprint ARG)`.

readint Reads an integer from input, stopping when it sees a newline. Has the syntax `(readint)`.

With the exception of `None`, all keywords are value-less (unless invoked with parentheses), and therefore cannot appear as operands to keywords, or to function calls.

3.3 Scope and Duration

Variables enter scope when they are first assigned to, and leave scope when you leave the function in which they were defined. Variables which are not declared within a function never leave scope. Naturally, variables declared outside of functions are out of scope until they enter scope, even if they stay in scope for the remainder of the program. It is illegal to use a variable (other than to assign to it) when it is out of scope.

These scope rules make it possible for different variables with the same name to be in scope at the same time when function argument names overlap with the names of top-level variables. In such cases the variable used should be the one declared within the function.

Conditionally executed declarations (i.e. first assignments which occur inside of if statements, while loops, or for loops, or the second operand of a short-circuiting operator) can be tricky to deal with, since we can't tell at compile time which branch of execution will be taken. Traditional solutions to this problem conservatively reject code which cannot be guaranteed to conform to requirements. We will do something similar and simply disallow conditional declarations – no variables may be declared inside of if-statement clauses, while loop bodies, for loop bodies, the second operand of short-circuiting operators, or for loop increments.

The rules of data duration are similar to those of scope – data continues to exist until such time as the function in which it is created returns, or until the end of the program in the case of data produced in the top level of the program. The exception to this rule is strings, which are not deallocated until the end of the program, no matter where they appear. Put another way, all data in LIFC has either static or automatic duration with integers and structs having automatic duration if allocated inside of a function and static if allocated outside of a function, and with strings always having static duration. If a value is accessed after the end of its lifetime then the resulting behavior is undefined.

3.4 Order of Operations

Because we are allowing for expressions which modify state it is important that subexpressions be evaluated in the proper order. The usual rule of thumb is that operands should be evaluated depth first from left to right, but there are a few special cases:

and,or Evaluate from left to right, but also short circuit when possible.

while Evaluate from left to right, but do so repeatedly until the predicate fails.

for Evaluate `INIT` first, then repeatedly execute `PREDICATE`, `BODY`, `INCREMENT` until the predicate fails.

4 Output

4.1 Correct Input

When given a valid LIFC program your compiler should produce an equivalent MAL MIPS program on `stdout`, produce no output on `stderr`, and return with exit status 0. You must also free all dynamically allocated memory prior to terminating your program.

There are no further restrictions on how you format your output. You can autogenerate comments for yourself, or not. You can follow the standard MIPS calling conventions we use in class, or not. You can write an optimizing compiler which attempts to minimize memory transactions, or not. The list goes on, but the bottom line is that as long as MARS produces the correct output when handed your program, and your program actually computes values, all is well.

For example, suppose you start with the following simple LIFC program:

```
(assign n 3)
(function (fib n)
          (if (and n (- n 1))
              (+ (fib (- n 1)) (fib (- n 2)))
              n))
(intprint (fib 6))
```

Then you might get the following output, assuming you spent far more time on this project than we expect you to:

```
.text
    li $a0 6
    jal fib
    #print $v0
    move $a0 $v0
    li $v0 1
    syscall
    #exit
    li $v0 10
    syscall
fib:
    beq $a0 $0 reta0
    addiu $t0 $a0 -1
    beq $t0 $0 reta0
    addiu $a0 $a0 -1
    addiu $sp $sp -8
    sw $ra 0($sp)
    sw $a0 4($sp)
    jal fib
    lw $a0 4($sp)
    sw $v0 4($sp)
    addiu $a0 $a0 -1
    jal fib
    lw $t0 4($sp)
    addu $v0 $v0 $t0
    lw $ra 0($sp)
    addiu $sp $sp 8
    j done
reta0:
    move $v0 $a0
done:
    jr $ra
```

4.2 Erroneous Input

In the case of erroneous input you should print an error message to `stderr` and return a nonzero exit status. We won't be terribly picky about the nature of the error message, but we do require that there be *something* printed to `stderr`, and that you exit gracefully, such as by a call to `exit()` as opposed to a segfault. You are allowed to "leak"[4] memory in response to garbage input.

5 Submission Instructions

In order to submit this project tag the commit you want to submit with the tag `proj1` and push to your git repo. For example, the following commands would submit the most recent commit in your local repository

```
$ git status #check that everything I wanted committed was committed
$ git tag proj1
$ git push --tags origin master
```

Don't be afraid about submitting multiple times – you can move your tags around as often as you please, so you can choose an older commit later on if you want, or a newer commit.

There is a file named `proj1.marker` at the top of the skeleton. We will be using that file to locate your submission when grading it, so be sure not to move it. If you're worried that we won't be able to find your submission properly you are also encouraged to use the `check_submission.py` script provided in the skeleton to sanity check yourself.

We'll be testing your submissions on the hive machines (the computers in 330 SODA, not the computers in 200 SDH). In addition to testing that your submission is in the right place on GitHub, be sure that your code works in the environment in which it will be tested. "It worked on my home machine" is not a valid excuse for a broken program, and will earn you no sympathy. Additionally, we'll be grading your code using an autograder (we may check a few things by hand, such as that you actually implemented your solution in C). That means you need to get things to actually work, not to sort of work – 5 test cases working is better than 25 test cases almost working, and of course if your code doesn't compile then it doesn't get credit.

[4]We use quotation marks here because while valgrind will think you've leaked memory, all allocated memory is freed when the process itself dies.

6 Tips

1. Start early. This project is an absolute monster. You *will* regret starting the day before it's due.
2. You can write tests at any time. Don't wait until you've implemented everything in your project to start testing, do it right from the beginning.
3. Write tests of varying complexity. Use simple tests that test out one functionality at a time to verify the soundness of each of the components of your compiler, as well as to help you debug. Use larger, more complex tests to check that everything works properly together.
4. Comment as you code. This will significantly cut down on the amount of time you spend scratching your head wondering what on earth you were thinking.
5. Don't be afraid to ask us for help. There's no point in wasting time burning rubber when we can help you get some traction.
6. Work incrementally. Try to get a few simple cases like integer literals and printing working first, then move on to more difficult tasks one by one.
7. Use your tools. `gdb` is usually a better debugging option than `printf` statements. Make use of your IDE's features as much as possible. You don't need to use the editors recommended by real programmers[5], but you should try to get to the point where you almost never need to use the mouse with your editor of choice. We're not making you use `git` purely out of spite – it should make your life easier, not harder. Commit frequently, and don't be afraid to start a new development branch if you have an idea you're not entirely sure of.
8. The way you write assembly by hand is not the same way that you want your compiler to write code (unless you don't feel like sleeping much). Try to do things so that every node you translate is independent of other nodes being translated.
9. There are a number of characters that we don't allow in names. In particular, `'$'` can appear in labels in MARS, but is disallowed in LIFC programs.
10. Depth first traversals are your best friend for this project.

7 A Brief Introduction to Compilers

This project is simple enough that many of the techniques used in the development of compilers would be ridiculous overkill, but the skeleton we've provided does break the process of compiling down into standard subcomputations of lexing, parsing, static analysis, and code generation. If you are having a hard time understanding what the skeleton is doing, then you may find this section helpful.

7.1 Lexing

The first step of compilation is lexing. In this stage we break a stream of input characters into a stream of semantically meaningful tokens. For example, the following LIFC program

```
(assign hello "Hello World!\n")
```

might break up into tokens as follows:

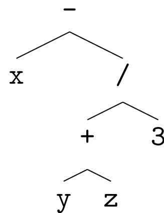
[5] `http://xkcd.com/378/`

```
OPEN_PAREN KEYWORD("assign") NAME("hello") STRING("Hello World!\n") CLOSE_PAREN
```

Normally this is accomplished by expressing the syntax of a language in Backus-Nauer Form (BNF), converting to a discrete finite automaton (DFA), and feeding inputs to the DFA. Fortunately LIFC is simple enough that we can instead greedily consume input until seeing whitespace or a parenthesis and emitting a token depending on types of characters consumed.

7.2 Parsing

The second step of compilation, parsing, takes the tokens generated by the lexer and builds them into a tree according to the semantics of the language. This tree is usually called an abstract syntax tree. For example, an expression like `x - (y + z) / 3` might generate an abstract syntax tree shaped like



More complex languages are usually parsed using BNFs and something called a shift-reduce parser, but once again, the structure of LIFC makes our life significantly easier than that. In particular, the syntax of LIFC is itself naturally tree-shaped. As a result, a simple recursive function which asks itself to create a subexpression tree whenever it sees an open parenthesis is a feasible solution.

Notice that because the top level of a LIFC program can contain multiple expressions that we are actually going to generate a list of abstract syntax trees (AST), rather than a single abstract syntax tree.

7.3 Static Analysis

This is the stage where static types, scope information, etc. are analyzed. Declaration gathering most properly belongs to static analysis, but was folded into the parser in the skeleton for the sake of convenience.

7.4 Code Generation

This is the final step in the translation process, and as its name suggests, this is where code is actually generated. Here our approach for this project is basically the same as the usual approach for more complex languages. For each different type of node in the AST we will define how it generates code. Whenever it needs to "evaluate" a subexpression it simply asks the corresponding AST to generate its code, recursively. This model is incredibly straightforward from a software engineering perspective, but it is important to make sure that the code generated by an expression "leaves everything where it found it", as this approach assumes a great deal of independence between different generated code blocks.

8 Change Log

2014-06-26: Clarified the allowed syntax of integer literals in section 3.1.

2014-07-05: Added rules for dealing with conditional declarations to section 3.3.

2014-07-09: Added short circuit-able operands to the list of places in which declarations may not appear (see 3.3).

2014-07-09: Made explicit the fact that keywords cannot be used as operands, with the exception of `None`.

2014-07-09: Made explicit the syntax for function calls.

2014-07-09: Clarified that function names follow the same rules as variable names.

2014-07-09: Added that the assignment operator can be paired with the arrow operator, as well as with variables. Because this feature was added late into the project we will not test it in the autograder.