CSCE 531
Spring 2016
Homework 1
Due Friday February 12 at 11:59pm

General Submission Guidelines

In general, all exercises that I assign throughout the course, except drawings, are to be submitted by CSE Dropbox (go to https://dropbox.cse.sc.edu). More detailed submission instructions are included with this handout. Drawings may, but are not required to, be submitted by dropbox as well. Alternatively, drawings may be submitted in class.

You must submit this assignment by the deadline above. No late submissions will be accepted for this assignment.

Assignment

Read this entire handout before starting the assignment.

This is an individual assignment. No teams.

This assignment has three objectives: (1) to get you used to writing C code with nontrivial data structures; (2) to introduce you to using flex; (3) to illustrate how a typical one-pass assembler works. You can access documentation for flex ("**f**ast **lex**") on the web or by typing "info flex" on any of the linux boxes in our labs (or logging in remotely).

You are to write a "baby" assembler/interpreter. Your program will read and internalize instructions written in a rudimentary assembly language (RAL for short), then simulate execution of the internally represented code. You have the option of making your flex program stand-alone, with all your code in one file, or you may distribute code among different C source files and/or C header files as well as the flex program. Your flex file name should end with ".l".

Submission guidelines for this assignment

Your program should be submitted as a gzipped tarball of a single directory named "hw1" which contains just your source code along with a Makefile to use for compiling your code (so that typing "make" will produce the executable). When you are ready to submit, remove all automatically generated files, go to the parent of your hw1 directory, and type

```
tar cvf hw1.tar hw1
gzip hw1.tar
```

This will produce a file named hw1.tar.gz which you then upload to CSE Dropbox.

Your executable should be called `ba` (for "baby assembler"). Typing "make clean" should remove any automatically generated files, including the executable. To run `ba`, you type

```
./ba <RAL code filename>
```

Here, `<RAL code filename>` is the name of the file containing a RAL program.

PLEASE NOTE: Unlike in Windows, file and command names in Linux are all case-sensitive. You must strictly adhere to the names above. Naming your executable "BA", your directory "HW1", or your submission file "HW1.tar.gz", for example, will get points taken off.

Except for flex, you must write your code in C. That is, program code with the flex program must be C code, not C++ or Java code, and the .c file that flex produces should be compiled with gcc, not g++ or any other compiler. There may be incompatibilities between default flex and C++, and you all should get more practice with C anyway. Be sure to test your programs thoroughly.

Your submission will be tested automatically, so you must adhere strictly to the interface above. More specifically, we will run a script on one of the linux lab machines (l-1d39-01.cse.sc.edu, for example) that will automatically compile and execute your program on various test files, capturing the output for comparison with that of the solution. If a glitch causes us to have to manually run your program, you will lose points.

We will provide a script that you can use to test your solution yourself just as we test it. Stay tuned.

Basics

Your two tasks---assembling a RAL source program and simulating its execution---are performed sequentially: Phase 1 is assembly; Phase 2 is running the internalized instructions. For Phase 1, the RAL source code will be read from a file given on the command line, and any error messages will be written to standard error (stderr). For phase 2, all input will be from standard input (stdin), and all regular output will be to standard output (stdout).

Phase 1 of your program should do some initializations then call yylex() only once. In other words, the flex pattern-matching engine should be the main driver of your assembly, and *you are only allowed to make one pass through the source program* (hence the term, "one-pass assembler").

RAL Syntax

Before we start, let's get our terminology straight. For our purposes,

- by *whitespace* I mean any nonempty sequence of spaces and/or tabs, but no newline characters;
- an *identifier* is as described in class, that is, an alphabetic character (including underscore) followed by zero or more alphanumeric characters (including underscore);
- an *integer constant* is a nonempty string of digits (leading zeros are allowed and have no effect), optionally preceded by either "+" or "-" (if the "+" or "-" is absent, we say that the constant is *unsigned*).

The RAL program you will parse is a text file consisting of a number of lines, each ending with a newline character. Each line contains one of three possible things: (1) a label followed by a colon; (2) a memory allocation directive; or (3) an instruction.

A line of type (1) looks like this:

```
$foo:
```

Remember that this is on a line by itself. A label is defined to be an identifier (for example, "foo") following a "$". Labels are used to indicate branching destinations.

A line of type (2) may look like

```
        .alloc  some_var
```

or like

```
        .alloc  some_array, 50
```

The ".alloc" is exactly as written (this is known as an *assembler directive*); some_var and some_array are any identifiers. There must be whitespace between ".alloc" and the identifier, and there is optional whitespace before ".alloc". The identifier may optionally be followed by a comma, optional whitespace, then a positive, unsigned integer constant.

A line of type (3) (i.e., a RAL instruction) starts with optional whitespace, then an *instruction name* (described below) followed by one or two *arguments* (also described below), separated from the instruction name by whitespace. If there are two arguments, they are separated by a comma (followed by optional whitespace).

There are five types of arguments (called *addressing modes*):

1. a *register*: a lowercase "r" followed by one of the digits 0-7,
2. a *direct memory reference*: an identifier,
3. an *indirect memory reference*: a register surrounded by parentheses,
4. an *immediate constant*: an integer constant,
5. a label.

Immediate constants must be in the range -2^{15} = -32768 through 2^{15} = 32767, that is, they must fit into a `signed short` (two-bytes). It is up to you what you do with values out of this range (e.g., quit with an error message or just truncate them); none of the RAL source code used to test your homework will violate this range limit.

Instructions come in different types according to the arguments they require:

- Memory access: `load`, `loada`, `store`. These require two arguments: first a register, then either a direct or indirect memory reference.
- Register: `move`, `add`, `sub`, `mul`, `div`, `mod`, `cmp`. All these require two arguments, the first a register and the second either a register or an immediate constant.
- Branch: `b`, `blt`, `ble`, `bne`, `beq`, `bge`, `bgt`. These require a single label argument (the *branch destination*).
- Input/output: `read`, `write`. These both require a single register argument.

The RAL syntax guarantees that there is no naming ambiguity between variables and labels. One may use the same name for both a variable and a label in the same program with no conflict. It is entirely up to YOU, however, whether you want to allow a register name or instruction name to be used as a variable name or label. Your homework will not be tested on any RAL code that does this.

It is also up to you whether to allow whitespace at the end of a line or before a label/colon combination. No RAL source programs used for testing will have any such whitespace.

RAL Semantics

There are eight available registers, `r0` through `r7`, each able to hold one `long int`. Main memory is modeled as an array of 2^{16} = 65536 long ints (not bytes). The address of a memory location is just its index in the array. There are only two primitive data types during execution: long (memory contents and registers) and unsigned short (used for indices (pointers)). On our lab machines, a long data type takes eight bytes and an unsigned short takes two bytes. All you need to know is that an unsigned short is automatically coerced into a long when needed, with no overflow.

Internalized instructions will also be kept in an array with 65536 entries, where each entry takes four bytes and is formatted as follows:

- First byte: a *opcode* describing the name of the instruction (e.g., load, add, bge, etc.)
- Second byte: two parts:
    - bottom three bits: a constant indicating which register is used as the first argument
    - upper five bits: the type of the second argument, if any

  (this byte is not used for branch instructions).
- Last two bytes: used for the second argument, which can vary depending on the instruction---either a code for a register, an immediate constant, an index to main memory, or a branch destination used for branch instructions. (These bytes are not used for input/output instructions.)

The sizes of main memory and the instruction array are chosen so that an index into either array can be stored in an `unsigned short` (two bytes). The instruction array entry at index 0 should not be used; put the first instruction at index 1.

Memory allocation

Variable and array names are declared using the `.alloc` directive. Variables and arrays are associated with chunks of main memory in the order that they are allocated. The line

```
        .alloc  <name>, <size>
```

associates the lowest index of available memory with the name, and reserves size many entries in the array (so the next allocation is at name+size). If the size argument is missing, then it is assumed to be 1. There is no distinction between variables and arrays; variables are just arrays of size 1.

All the record-keeping for allocation takes place at assembly time, not runtime. There is no runtime ("dynamic") memory allocation; all variable and array names refer to static locations determined at assembly time. When a variable/array is assigned a memory location by an .alloc directive, all future uses of that variable/array name refer to that location. To support this, you must maintain a dictionary (called a *symbol table*) of name-location associations during assembly. Only the locations of variables are retained after assembly; the names are forgotten.

Instructions

Most instructions, when executed, alter the state of main memory or the registers as follows:

- `load`: Copy the *contents* at the memory location given by the 2nd argument (a value of type long) into the register given as the 1st argument. If the 2nd argument is an indirect reference, then the contents of the corresponding register are interpreted as an index into main memory (like a pointer), and the contents of memory at that index are copied.
- `loada`: Copy the *address* (i.e., the index in main memory) given by the 2nd argument into the register given by the 1st argument.
- `store`: Copy the contents of the first argument register into the memory location given by the second argument.
- `move`: Copy the contents of the 2nd argument (register or immediate constant) into the 1st argument (register).
- `add`: Add the 2nd argument to the 1st argument.
- `sub`: Subtract the 2nd argument from the 1st argument.
- `mul`: Multiply the 1st argument by the 2nd argument.
- `div`: Divide (integer quotient) the 1st argument by the 2nd argument.
- `mod`: The 1st argument gets the remainder upon dividing it by the 2nd argument.

Note that in all the register instructions, the first argument holds the result and the second argument (if it is a register) is unchanged.

The *comparison instruction* `cmp` compares the values of its two arguments and stores the result---either LESSTHAN, EQUAL, or GREATERTHAN---for use by the next branch instruction (see Control Flow, below).

Control flow

When a RAL program is run, instructions are normally executed in the order they appear in the source, starting with the first instruction. The execution order can only be altered by a branch instruction, whence, if the branch condition is met, the next instruction executed is the one immediately following the type (1) line containing the branch destination label. This altered flow is called a *branch* or *jump*.

The branch condition is determined by the branch instruction and the result of the most recently executed `cmp` instruction. For example, in the code

```
        load    r0, 17
        load    r1, 18
        cmp     r0, r1
        blt     $hop_over
        add     r0, r1
        cmp     r0, r1
        bge     $byebye
$hop_over:
        sub     r1, r0
$byebye:
        ...
```

the branch to $hop_over is taken by the first `blt` instruction, because the result of the previous comparison was LESSTHAN. If instead, the first two instructions were

```
        load    r0, 18
        load    r1, 17
```

then the first branch would not be taken, and control would continue normally to the `add` instruction. (The second branch would be taken in this case.)

The branch is taken based on the result of the last comparison as follows:

- LESSTHAN: take branch with `b`, `blt`, `ble`, `bne`
- EQUAL: take branch with `b`, `ble`, `beq`, `bge`
- GREATERTHAN: take branch with `b`, `bne`, `bge`, `bgt`

Note that `b` executes an unconditional branch.

The branch instruction need not follow the comparison immediately. There may be several non-comparison instructions in between.

Execution finishes when control falls past the last instruction in the program.

Input and output

The `write` instruction outputs the contents of its register argument (as a signed long int in decimal) to stdout, followed by a newline character. No special output formatting is done.

The `read` instruction reads from stdin a (possibly signed) integer constant, ignoring any whitespace or newlines preceding it. You can call the C library function `scanf` to accomplish this. A C++ equivalent is

```
    cin >> some_location_of_type_long
```

A sample RAL program

Here is a little program that reads in 20 numbers from input and outputs them in reverse order:

```
        .alloc  numlist, 10
        loada   r0, numlist
        move    r1, r0
        add     r1, 20
$read_loop:
        cmp     r0, r1
        bge     $write_loop
        read    r2
        store   r2, (r0)
        add     r0, 1
        b       $read_loop
$write_loop:
        loada   r1, numlist
        cmp     r0, r1
        ble     $quit
        sub     r0, 1
        load    r2, (r0)
        write   r2
        b       $write_loop
$quit:
```

Backpatching

As you assemble, you maintain a dictionary of label-location pairs, so that you can translate a label used as a branch destination into the actual destination (as an index into the array of instructions). This is fine as long as the label has already appeared followed by a colon on a line by itself. But this may not be the case; a branch may be forward to a destination further down in the code, as in the example above (the first branch to $write_loop and the branch to $quit). In fact, you may have several forward branches to the same label. Because you can only make one pass on the input, you might not know the destination of a branch when it appears. For each such unresolved label, you must keep a list of branch instructions that use that label as a forward branch destination. When (or if) the label is resolved, you then go back and fill in the correct destination into those branch instructions.

A particularly space-efficient way of handling forward branches is by embedding the (linked) list of unresolved forward branch instructions in the branch instructions themselves. This technique is known as *backpatching*. When you first encounter a forward branch to some label $foo, you enter $foo as a new key into the dictionary of unresolved labels with the index of the current branch instruction, say, *x*, as its value. You also enter enter 0 for the branch destination of instruction *x* (recall that 0 is not the index of any instruction). The next time you encounter $foo as a forward branch in some instruction *y* > *x*, you place *x* as the branch destination in instruction *y* then update the value of $*foo* in the dictionary from *x* to *y*. You continue this process for each subsequent occurrence of $foo as a forward branch: place the location of the previous forward branch to $foo in the current instruction and update the dictionary with the current instruction location. When $foo is resolved, you have a linked list of its forward branches embedded in the instructions themselves, which you can then resolve one by one. I will also discuss this in class.

Error checking

You need to check for the following semantic errors:

1. A varable or array name must be introduced by an .alloc directive in the source code above where it is used as an argument.
2. The same name cannot be used in more than one .alloc directive.
3. No memory allocation may exceed the bounds of the memory array.
4. Any label used as a branch destination must occur exactly once on a line by itself, followed by a colon. The latter need not occur before the label's first use as a branch destination, however (see Backpatching, above). That is, all labels must have been resolved to unique instruction indices when assembly ends.

When you encounter one of these errors, you must issue some appropriate error message to stderr and keep assembling, but you do not proceed to Phase 2 (execution). There will be no other types of errors in any RAL sources used to test your code, so what you do upon encountering some other error (e.g., a syntax error) is entirely up to you, including promptly quitting the program.

You need not check for any run-time errors. There will be none in the sources we use to test your homework (we hope!).

Testing

I will post test files on Dropbox in the days to come, along with the grading script. In the mean time, you should do your own testing with simple RAL programs.

When done testing your program, type "make clean" to remove automatically generated files before you submit.

Efficiency requirements

There are no specific efficiency requirements for this assignment. You should exercise common sense and make your code reasonably efficient, both in time and space. You might consider implementing your symbol table as a hash table with chaining (rather than open addressing), but this is not required. You should, however, use *some* linked structure, because there will be no a priori limit on the number of symbols (variables or labels) introduced in a source program.

Features of flex you cannot use

I am disallowing anything in flex beyond its default features described in class. In particular, do not use any flex options such as start conditions, states, REJECT features, etc. The reason is not because these options are bad. In fact, they can be very powerful -- a bit too powerful. Instead, I want you to focus only on the core features of flex. Restricting yourself to the default, core behavior will make your code more readable and portable.

Miscellaneous hints

Flex rules

The rules you use for pattern matching are somewhat flexible (no pun intended). You could use a single rule to capture each line of source in its entirety, such as

```
.*\n      do_something_with(yytext);
```

But then you have to parse each line "by hand," so you are not really using flex to anywhere near its full potential, and you are missing the whole point of using flex in the first place. Let flex do the parsing work for you!

You could instead have several different rules that match different types of source lines in their entirety. In that case, flex tells you what kind of line it is: allocation directive, label, or instruction, and if so, what kind of instruction, etc. Then you still have to go in and parse the line internally by hand to get at the arguments. Alternatively, you could have rules for the various components of a line, e.g., label/colon, ".alloc", the various instruction names, each individual argument (with a separate rule for each type of argument). Then you would keep track of your progress through a line by maintaining various global flags. There are advantages and disadvantages to both approaches, but on balance, I think the latter approach is better; it allows flex to pull out the minute details of the source.

Handling strings in C

In C, a string is just an array of characters, ending with the null character (ASCII 0). There is no special string type. In flex, the text of a match is stored in the array `yytext`, but this array is clobbered by the next match. If you want to store the match for safe keeping, you'll need to copy it to a new location. For example,

```
    #include <string.h>
    /* ... */
    char *new_str;
    /* ... */
    new_str = (char *) malloc(strlen(yytext)+1);
    strcpy(new_str, yytext);
```

You need the "+1" to make room for the null character (ASCII value 0) terminating the copied string. The integer value `yyleng` is also set to the length of the match, so instead of `strlen(yytext)` you could have just used `yyleng`.

To compare two strings lexicographically (which includes checking if they are equal), use `strcmp()`.

More hints (answers to FAQs)

Any code you see below you are free to copy and modify without attribution.

Code for a hash table

Skip this if you are using something besides a hash table with linking for your dictionary.

Here is how I define my hash table:

```
#include <stdlib.h>
/* ... */
#define INITIAL_HASH_SIZE 8
#define MAX_LOAD_FACTOR 2
#define SCALE_FACTOR 2
/* ... */
int h_size;      // Current size of the hash table array
int num_items;   // Number of items stored in the dictionary
DR *hash_tab;    // The actual array (dynamically allocated)
/* ... */
h_size = INITIAL_HASH_SIZE;
/* ... */
/* Assume DR is a type defined as a pointer to a hash table record */
hash_tab = (DR *) malloc(h_size * sizeof(DR));
```

Here is my home-grown, general-purpose hash function:

```
int hash(const char *key, int h_size)
{
    int sum = 0;
    for (; *key; key++)
        sum = (37*sum + *key) % h_size;
    return sum;
}
```

There is nothing special about 37, except that it is prime. For performance reasons, h_size should not have any factors in common with 37, which means that, because 37 is prime, it should just not be a multiple of 37.

When the load factor (the ratio of num_items to h_size) exceeds MAX_LOAD_FACTOR, I resize the hash table up by a factor of SCALE_FACTOR. That is, I allocate a new array with size h_size*SCALE_FACTOR (updating h_size accordingly), move all records from the old array into the new array (recomputing each hash value in the process), and de-allocate the old array (free(old_array)). These numbers can vary, but SCALE_FACTOR should be at least 2, and MAX_LOAD_FACTOR should not be more than, say, 5 or 6 or so.

Line numbers

You are not required to keep track of line numbers, but if you want to (for error messages, say), you can do this easily in flex by including a rule for the newline character "\n", whose action increments a global count.

Standard output versus standard error

This assumes you have the #include <stdio.h> directive somewhere.

Every C program is given two output streams: standard output (stdout) and standard error (stderr). Normal output should be sent to stdout, while any warnings or errors should be sent to stderr. Without redirection, both streams are echoed to the terminal, so they appear interleaved with each other.

But beware! These two streams are handled differently and are not synchronized. Stdout is buffered while stderr is not. That means that stderr messages go out immediately, while stdout output may be delayed in a buffer. This means that the actual order that things appear on the terminal does not necessarily reflect the order that they are generated by the program.

If you want to better synchronize things, you can flush the stdout buffer at any time by calling fflush(stdout). For example, if you do this just before sending an error message to stderr, then your normal output will appear on the terminal before the error message.

Do not send warning or debugging messages to stdout! If you do this, the warnings will be mixed in with the normal output, and so the output files won't match. Instead, send the messages to standard error using

```
fprintf(stderr, "Warning: ...", ...)
```

instead of

```
printf("Warning: ...", ...)
```

which sends the message to stdout. fprintf() is called just like printf(), but takes the stream to send the output as the first argument.

Debugging

I don't use any specialized debugging tools when I write code, so I can't help you with, e.g., gdb. Instead, I include output statements in my code in various places as needed. I always use stderr for the output stream for debugging messages, because (1) they won't get mixed up with normal output, and (2) since stderr is unbuffered, the debug messages won't be lost in a buffer if the program crashes.

DOS newlines versus Unix/Linux newlines

In the Windows world, lines in text files end with the sequence \r\n (carriage-return, linefeed), whereas in Unix/Linux they end in \n only. Please be aware of this if you edit text files on Windows. You can hand-edit your files with emacs on Unix/Linux to remove the offending carriage returns, or you can run your file through the lexer generated by this flex program:

```
%%
\r\n putchar('\n');
%%
int main()
{
        yylex();
        return 0;
    }
```

(You compile lex.yy.c using the -lfl switch.)

The od (octal dump) command

This is a great command! Use it to inspect the contents of a file. For example,

```
    od -c mytextfile
```

lists each byte of the text file in a readable way. This is especially useful to determine the presence and nature of whitespace in your file, which is often hard to do when viewing your file in some editors. Other switches give other output formats. Type

```
    man od
```

for more information.

Getting started

Below are just my recommendations. Your mileage may vary.

First, make sure you are clear on what the sample code given above does and how.

Implement the flex pattern matcher next, because this is probably the least familiar to you. To start with, you can have trivial actions that maybe just print the texts of the matches. This will allow you to test whether you've got your patterns correct.

Next, implement the dictionary(ies), including routines for searching for, adding, and updating an entry, as well as marking unresolved labels. Please, please, PLEASE make every effort to implement this data structure from scratch. That way, you will learn the C programming techniques that are most important for the main project to come.

Flex examples you can run and modify

To help you get more of a feeling for using flex, I have provided two sample flex programs, with Makefiles, used in previous classes in the subdirectories "wordcount" and "tokens" (download and unzip this file). The latter was taken directly from the flex tutorial (see "info flex" above). Copy each directory to one that you own. For the tokens program, type "make", then

```
./tokens sample.pas
```

and observe the output. For the wordcount program, type "make", then

```
./wordcount
```

then type a few lines of text, ending with ctrl-D on a line by itself (ctrl-D provides an end-of-file marker for keyboard input).

When done testing, type "make clean" to remove automatically generated files before you submit.

You are free to adapt and modify either of these programs for your homework without citation.

More FAQs

1. *My symbol table is screwed up. It looks like some keys have garbage in them. Why does that happen?* Most likely, you directly stored the pointer value `yytext` directly in a dictionary record without copying the string to a permanent, dynamically allocated location. The buffer that yytext points to should be considered volatile---it is altered with every match.
2. *What is the `next` field in a dictionary record for?* It is used to point to the next record that hashes to that entry in the hash table. It is part of the standard way to implement a linked hash table and has nothing to do with key-value pairings.

This page was last modified Thursday January 28, 2016 at 10:49:42 EST.