Week 11 assignment: Compiling Jack

1 Tasks

1. Extend the given skeleton into a full Jack parser.
2. Extend this parser into a compiler from Jack into Hack VM.
3. (Optional!) Combine this with your Hack VM translator and Hack assembler to create a full compiler from Jack into Hack machine code.

2 Required software

For this lab, you will need some way of comparing two text files for differences. One way is to use the "fc" terminal command on Windows or the "diff" terminal command on Linux and Mac OS. On non-lab non-Mac machines, you can also use e.g. Meld, a piece of free open-source software with a nice graphical user interface.

3 Lexing and parsing with tag.c and tag.h

While lexing, we will store tokens in files as XML tag pairs `<tokenType> tokenContents </tokenType>`. For example, the `keyword` token `else` is stored as `<keyword> else </keyword>`. The first tag, `<keyword>`, is called the *opening* tag. The second tag, `</keyword>`, is called the *closing* tag. All tag pairs in XML must be of this format, with tag names enclosed by `<>`s and with the closing tag starting with a `/`. Everything between the two tags is referred to as the *content* — in this example, the content is *else*. The <, >, `&` and `"` symbols cannot be stored directly as tag contents and must be escaped to `&gt;`, `&lt;`, `&amp;` and `&quot;` respectively.

To facilitate all this, we have modified `token.c` and `token.h` from weeks 8–10 into `tag.c` and `tag.h`. These files provide the following facilities:

- A `Tag` struct. Similarly to tokens, a `Tag` contains a type (which is an enum `JackTagType`) and `data` (which is a union `TagData` and stores the contents).
- Functions `malloc_tag` and `free_tag` to create and dispose of `Tag`s in a memory-safe way.
- Functions `read_tag` and `write_tag` to read tokens from and write tokens to a file as XML tag pairs.

In parsing, we will store non-terminal elements in XML format. All children of the non-terminal in the parse tree appear as contents of the tag, as seen in video 11-2. `tag.c` and `tag.h` support reading and writing open and close tags of non-terminal elements separately as follows:

- Non-terminal open and close tags have a special `JackTagType`, namely `NON_TERMINAL`.
- The `data` of a non-terminal open or close tag is an enum `NonTerminal` which specifies the type of non-terminal.
- A `Tag` has a `close_tag` field which is true for a closing non-terminal tag, false for an opening non-terminal tag, and ignored for a token's tag pair.
- The functions `read_tag` and `write_tag` both work for non-terminals in the way you'd expect. They also indent the file in a smart way.

We have also provided some helper functions that will be particularly useful in parsing with `Tag` pointers `current` and `lookahead` as in video 11-2:

- `advance_tag` reads the next token from the input file, advancing `current` and `lookahead` accordingly
- `copy_tag` copies `current` from the input file to the output file, then calls `advance_tag`.
- `write_non_terminal` writes an opening or closing XML tag for the given non-terminal to the output file without changing the value of `current` or `lookahead`. It's really just a wrapper for `write_tag` that handles creating the `Tag` instance from the given data.

4 Part 1: Parsing Jack

The first part of getting a working Jack compiler is lexing and parsing a `.jack` file into a `.vm` file. The first command line argument should be the input file, and second argument should be the output file. We've provided a code skeleton with the following functionality:

- `main` first lexes the file into output `lex_out.xml` using the (fully-implemented) function `lex_file`, then calls the (partially-implemented) `parse_file` function with a file input of `lex_out.xml` and an output of `parse_out.xml`. Finally, it calls the (unimplemented) `compile_file` function with an input of `parse_out.xml` to generate the final output.
- `parse_file` starts by setting up `current` and `lookahead` as in video 11-2. Annoyingly, the list of tokens should start with a special `<tokens>` tag to conform to the XML standards — `parse_file` skips this with a call to `advance_tag`. A `.jack` file should consist of a single ⟨class⟩, so it then calls the `parse_class` function. On return, the entire token file should be parsed and `current` should be pointing to the special `</tokens>` tag at the end, so `parse_file` handles cleanup and closes.
- For the parser as described in video 11-2, each non-terminal should have a dedicated `parse_` function. We have provided the `parse_class` and `parse_class_var_dec` functions as worked examples.

You will need to create `parse_***` functions for the other non-terminals in the grammar of video 11-2 (except for ⟨type⟩ as described in that video). Each function should have arguments of `current`, `lookahead`, `input` and `output` as in the headers provided. You may find the following tips useful:

- You do **not** need to carry out careful (or indeed any!) error checking on the provided Jack code.
- As described in video 11-2, you'll probably only need to use the value of `lookahead` once, inside the `parse_term` function.
- In order to match the test data exactly, you will need to suppress the ⟨type⟩ **and ⟨parameterList⟩** non-terminals from your output (i.e. don't write open or close tags for it to the output). You will also need to suppress the ⟨subroutineDec⟩ non-terminal; that said, we strongly recommend you do have a `parse_subroutine_dec` function, as having `<subroutineDec>` and `</subroutineDec>` tags will be useful in code generation.[1]
- If you're unsure about whether or not something is valid Jack, while your first port of call should be the official grammar, you can always fire up the existing Jack compiler to check! It's available for download from the nand2tetris site, or it's now included in the bundle of software from the unit page.

5 Testing your parser

You'll be using the test scripts from the original nand2tetris course (available from the unit page). These are as follows:

- `ExpressionLessSquare` is a nonsense Jack program in which every expression has been replaced with a single identifier, and there are no array subscripts.
- `Square` is the "real" version of `ExpressionLessSquare`. It's discussed in more depth in chapter 9 of Nisan and Schocken, and is essentially a Jack version of the Rogue exercise from week 5. It does not require any array subscripts.
- `ArrayTest` is the example program used in video 11-1, using both expressions and array subscripts.

All three tests come with come with two XML files per Jack file: `foo.xml` and `fooT.xml`. In each case, `foo.xml` is the desired output of your parser when run on `foo.jack`, and `fooT.xml` is the desired output of your lexer. In each case, you should run your parser on all files `foo.jack` and compare the outputs to the corresponding files `foo.xml` using fc, diff, or Meld. You hopefully won't need `fooT.xml`, but we've included it to check against our lexer output in case you want to rule out bugs there.

[1] Basically, because the original nand2tetris assignment assumes students will be working in a more pleasant language than C, they leave people to implement the parser from scratch rather than using a comparatively robust framework like this one. Putting the `<subroutineDec>` tags in the right place is tricky without a robust framework, so they recommend people don't bother. For us this shouldn't be an issue.

6 Part 2: Semantic analysis and code generation

In this part you'll be extending your Jack parser into a full compiler. In `symboltable.c` and `symboltable.h`, we've extended the symbol tables from the week 8 assignment (the Hack assembler) with the following functionality:

- The `TableEntry` struct now contains `type` and `kind` field in addition to the `name` field, and the old `address` field has been renamed to `offset`. `type` is a string. `kind` is an enum of type `VariableKind`, and distinguishes between local, argument, field and static variables. The basic functionality remains the same — it's just a data container.
- The functions `malloc_table`, `free_table`, and `get_table_entry` functions are unchanged.
- The `add_to_table` now requires a type and a kind in addition to a name. In addition, the auto-generated `offset` for a table entry now only increments for other entries of the same `kind`. For example, adding an entry of kind `VK_VAR` to a table with three `VK_VAR` entries and ten `VK_ARGUMENT` entries will give it an offset of 3 (as the fourth `VK_VAR` entry), not 14 (as the fourteenth entry).
- The new function `is_primitive` returns true if the given table entry is of type `int`, `char` or `boolean`.

We've also provided a `CompileData` struct to reduce the number of arguments that need to be passed around, which is initialised in the `compile_file` function provided. We have also given you the `compile_class` function to get you started. You should write the rest of the `compile_***` functions as described in the videos this week — we've provided suggested function headers and functionality for each one. You may find the following tips useful:

- Remember that the Hack character set agrees with ASCII except on special characters. This means that when compiling string literals, you can cast from `char` to `int` or use `sprintf` rather than needing to write a lookup table with 50+ entries.
- In the VM emulator, when the emulator is set to "No animation" for fast running, the animation speed slider controls clock speed. This is extremely helpful for test scripts like `Square` or `Pong`.
- In the VM Emulator, you can set breakpoints for every time the code enters a given function using the "currentFunction" variable. This is an easy way of skipping past calls to library functions (which will always work correctly).
- In the VM emulator, if your program crashes, you can see the call stack with a list of all the functions called in order at the bottom left of the window.
- In general, if the Jack code provided isn't compiling correctly, don't be afraid to write your own Jack code with a "minimal failing example" and get that working in the VM emulator first.

7 Testing your compiler

You'll again be using the test scripts from the original nand2tetris course (available from the unit page). After compiling, run each one in the VM emulator, making sure to load the whole folder, and check their behaviour is correct. (This also contains the standard libraries, which many of them use.) The scripts are as follows:

- `Seven` just prints 7 to the screen.
- `ConvertToBin` converts the value in RAM[8000] to binary and outputs it to RAM[8001]–RAM[8016], **from least significant bit to most significant bit** (so it will appear in reverse order in the memory viewer). Remember you can use the binoculars icon in the VM emulator to quickly jump to a given memory address.
- `Square` draws a square on the screen, which you can control with the arrow keys. You can increase the square size by pressing 'z' and decrease it by pressing 'x' **as long as the square is either moving or in the top-left corner** — otherwise, nothing will happen. Pressing the 'q' key will "quit", which doesn't visibly alter the screen but does prevent any response to further input.

- `Average` is the program from the first video this week with some minor changes. If you're having trouble debugging this one, I strongly recommend reducing the lengths of the string literals in the Keyboard.readInt calls to make it faster to get back to the point that's having issues.
- `Pong` is a one-player implementation of Pong, complete with a score (one point per hit), a bat that shrinks with each point, and a game over screen.
- `ComplexArrays` runs some tests on tricky array expressions and prints both the expected and the actual results — these should match.