JSTCC: A simple C Compiler
CS660: Compiler Construction

Janelle Blankenburg, Shubham Gogna, Terence Henriod

December 16, 2015

Abstract

Over the course of the Fall 2015 semester we implemented a basic compiler for the C programming language. The compiler is composed of a front-end and a back-end, which interface through a psuedo-assembly language of our own design (referred to as "3 Address Code" or "3AC"). The compiler translates ANSI C to MIPS assembly that can be run using the Mips Assembly and Runtime Simulator (MARS).

1 Compiler Construction Course

The goal of the course was to implement a basic, ANSI C compiler, oriented toward learning compiler concepts and not necessarily something "production ready."

Over the course of the semester, a compiler that uses a front-end → intermediate representation → machine code methodology was used. The compiler was constructed in stages, such that an end-to-end compilation was not achieved until the end, requiring better initial design of components than might have been required otherwise.

The product of the course took the form of `jstcc.py`, a python script embodying the compiler. The compiler was named for the members of the group: the **J**anelle **S**hubham **T**erence **C** **C**ompiler.

2 Implementation

2.1 Overview

The compiler was implemented in the Python3 programming language in order to emphasize compiler concepts and code readability. A few third party libraries are used, but the majority of the code and design is our own.

The team consisted of two graduate students (Janelle, Terence) and one undergraduate (Shubham), and completed the graduate level requirements of the course.

The compiler is composed of the following steps or modules, which will be described in detail later:

1. Symbol Table
2. Scanner/Tokenizer
3. Parser/Syntax-Directed Translator
4. Abstract Syntax Tree (AST)
5. Three Address Code (3AC)
6. Machine Code (MIPS Assembly language)

2.2 Dependencies

The following 3^{rd} party libraries were used:

- **Python3 Interpreter**: Necessary for running Python code.
- **bintrees**: A pure python balanced-tree data structure library. Used to meet the requirement of using balanced trees in the symbol table.
- **PLY**: Python Lex-Yacc. A pure python, tried and true Lex and Yacc implementation. these were used to implement the scanner/tokenizer and the parser or syntax-directed translator.
- **pylru**: A python LRU cache implementation used to implement (in part) register management in assembler output.

2.3 Usage

2.3.1 JSTCC

In order to run our compiler, you must have the above dependencies installed. Then you can run the compiler from the command line with a command like:
`$ python relative/path/to/JST/project/bin/jstcc.py <flags> <c program file>`

JSTCC has several command line options to produce or add various items to its output:

- `-h`, `--help`: Displays usage information for the program.
- `-o`, `--outfile`: Designate a `.asm` file to place the mips output in (default: `stdout`).
- `-sym`, `--symtable` Add a dump of the symbol table to the output.
- `-s`, `--scandebug`: Add printing of tokens and source code to the output during tokenization.
- `-p`, `--parsedebug`: Add printing of productions and source code to the output during parsing.
- `-ast`, `--astree`: Add the textual representation of the tree in the GraphViz language to the output.
- `-tac`, `--threeac`: Add 3AC and source code to the output.
- `-mips`, `--mips`: Add 3AC and source code to the generated `.asm` file.
- `-w`, `--warnings`: Activate printing of compile-time warnings to the `stdout`.

2.3.2 Web-Site

Our website features presentations of our project through its various stages, including presentation forms of our most important test cases and their output, output including informative intermediate representations, MIPS assembly, and the output of running MIPS programs in MARS.

To access the web-page statically, simply open the `index.html` file with a web browser (the operating system often has an association between this type of file and the web-browser, so opening the file should work).

2.4 Symbol Table

The symbol table was used to contain information about all of the declared items in the program (variables, functions).

In the beginning of the project, the symbol table played a central role in the organization of the compiler's information. However, as the project went on, the symbol table's use became reduced to almost just tracking memory usage and function signatures.

2.5 Scanning and Parsing

PLY was used to implement the Scanner and Parser. This allowed us to implement this items in an easily extensible way.

For the scanner, much of the difficulty came in finding the correct regular expressions to describe the tokens properly, but in the end this was very doable.

As for the parser, it was a harsh realization that it played an extremely large role in the project, had we realized this sooner, the project would have gone much better. Getting accustomed to thinking in the way of an LR parser was also difficult, but once we did, things progressed. This is an area of the project that we think we could really do much better with had we known what it would entail ("If I knew what I know now

when I was younger").

2.6 Abstract Syntax Tree

The AST was a real turning point for the group in terms of the project. This was the point when we started to glimpse "the big picture."

Designing the tree nodes was straightforward for the most part, but when it came to designing nodes like the Symbol node, there was really no good way to design it properly without knowing what we would learn later in the semester. Getting a good Symbol node and an excellent type system would be the areas that it would have been most effective to re-implement given the chance.

Our AST used relatively few nodes, and we aimed to keep a set of nodes that eschewed anything extraneous and would not result in intermediate code, but not have a set so minimal that the nodes were no longer modular. Further, we feel our design would be easily extensible given that the nodes were designed to interface with one another in such a way that they would not block another node's functionality.

2.7 Three Address Code

Our psuedo-assembly 3AC was closely modeled after MIPS. In retrospect, it would have been nice to build something more general, but since we were building a machine to translate to MIPS, this was acceptable.

The 3AC was designed to accept registers, literals, and addresses as parameters, as opposed to types for arguments as others may have done, but we felt this greatly eased the transition to actual assembler code. The drawback, of course, is difficulty in use of the co-processor, but it certainly would not be prohibitive, and floating point operations could be implemented with little more effort than had we taken the alternative approach.

A note on implementation: we used a single, generic 3AC instruction class; if we could, we would revise this to use a unique class for each instruction, possibly inheriting from relevant base-classes to create categories of related instructions.

2.8 Assembler: MIPS

The implementation of the final translation to MIPS was both a relief and a pain. We finally were able to see our project come to fruition (since the compiler was built in such a way that it was difficult to know exactly how things would turn out, building in a cross-sectional instead of longitudinal way).

In order to implement the MIPS translation, several supporting frameworks had to be created. The first was a register use table, to help manage the mapping of psuedo-registers (used prolifically in the 3AC) to the limited physical registers required for MIPS programming. The other was The development of MIPS macros that could be inserted into any program, greatly reducing the footprint of any generated code.

The most difficult part of MIPS translation was the design and implementation of the stack/activation frame. This was complex and very error prone, partly due to the confusion that comes with the fact that *stacks grow down*. Also difficult was determining how to separate responsibilities of *caller*s and *callee*s. In the end, the result was as follows:

1. The caller pushes registers and a copy of the register-spill memory whose values need to be saved onto the stack.
2. The caller then copies/pushes the values of any function call arguments onto the stack.
3. The caller executes a jump-and-link instruction to the callee.

4. The callee then pushes memory onto the stack for all of its local variables.
5. The callee executes its body.
6. Upon completion of the function body or a return statement, the callee stores any relevant return value to the designated return register and de-allocates its local variable and argument memory.
7. The callee jumps back to the caller.
8. The caller then recovers the spill memory and register values.
9. Finally, the caller copies the return value from the designated return register for use.

A triumph in our register use was detecting points in the program when we no longer needed to maintain mappings for pseudo-registers to physical ones. This greatly economized our use of spill memory, often eliminating the need to spill registers altogether! This was very good considering that we limited ourselves to roughly half of the available MIPS registers

3 Compiler Capabilities

The following items were requirements of the course:

1. Basic declarations and assignments.
2. Simple arithmetic operations.
3. Conditional statements (`if`, `else if`, `else`).
4. `while` loops.
5. One, two, and three dimensional arrays.
6. Basic function calls with simple parameters.

Our compiler supports the following additional features:

1. `for` and `do... while` loops.
2. Arbitrarily high dimensional arrays.
3. Recursive function calls.
4. Functions that accept arrays as parameters.
5. Acceptance of "fragmented strings", where string fragments such as `"hello" "world" "!"` are combined into a single string literal.

Additionally, the following optimizations are supported:

1. Constant folding.

4 Future Work

The following are features that, if given more time, are features that we would like to or be able to implement. There are simple ones that would only take days to implement, and complex ones that would require more effort.

4.1 Simple

- *Flesh out floating point operations.* We made a start, but did not have time to see it through given that it was not a requirement of the course.
- *Make use of floating point and string literals.* We have a good concept of how to do this, but since it was not a requirement we did not quite make it there.
- *More minor optimizations.* We would like to implement other simple optimizations such as replacing multiplications or divisions by two with bit-shifting operations.
- *Implement pointers.* We were very close in thought and effort to accomplishing this, but again, since it was not a requirement, we refocused our efforts elsewhere.

4.2 Complex

- *Enhanced error reporting.*
- *Flesh out all basic language features.* It would have been good to implement ternary operators and `switch` statements, but given their redundancy to other language features, they would be more for completeness and a bookkeeping exercise than an actual enhancement.
- `structs`. These are the next step in a complete language and implementing them would be very informative about memory management concepts.
- `typedefs`. For both the convenience and the "intellectual street cred."
- *Function pointers.* To truly master the language, create a language capable of true generality, and to better understand code manipulation.

5 Recommendations

We would first recommend to anyone building a compiler in the future to make use of a unit test framework. Unit testing as much of our code as possible, even if it was just making sure that objects resulting from processes could be stringified in a predictable manner or to evaluate if fatal errors occurred in the code was instrumental in our progress. Being able to test things in an automated manner saved many hours of strained eyes and tired fingers.

Second, we would recommend that compiler-implementing-students look to real-world examples as much as possible. We drew inspiration from a library called *pycparser*, a python C parsing library, for most of the front- end. This was helpful, but in retrospect, we wish we would have looked at an established intermediate language such as *LLVM* to model our 3AC after. Also it may have provided the benefit of having an already-implemented back-end to compare against.