Chris Altonji

Allen Chen

Andrew Smith

CS 4240 Phase III

Compiler repository: https://github.com/ClysmiC/tiger-compiler/tree/master

(You can find google-doc links to our table spreadsheets in our github's readme. They are much more readable than opening the raw .csv files in a spreadsheet program)

Project Structure

The project directory contains the following:

- `src/` : Contains all of the source code.
    - `com/tiger/compiler/` : Source code for the compiler as a whole.
        - `frontend/` : Source code for the front end of the compiler.
            - `irgeneration/` : Source code for the pass that generates IR code
            - `parser/` : Source code for the parser. The parser produces the symbol table, and also generates a parse tree of the program that is used for the semantic analysis and IR generation passes.
            - `scanner/` : Source code for the scanner.
            - `semanticanalysis/` : Source code for the semantic analysis pass of the compiler.
        - `backend/` : Source code for the back end of the compiler.
            - `assemblygeneration/` : Source code for the instruction selection process.
            - `registerallocation/` : Source code for the various register allocation schemes.
                - `cfg/` : Source code for constructing a control flow graph for the basic block registration scheme.
- `res/` : Contains the .csv files of the various tables and grammar rules needed to drive the scanner, parser, and symbol table construction.
- `test/` : Contains various test programs, as well as the output of running the compiler on them with the `-allc` flag. There are output assembly files for the `-r1` and `-r2` flags.

- `handwritten/` : hand-drawn DFA's for the scanner (Phase I), and an image of how the semantic stack (Phase II) is interpreted for different semantic actions.
- `grammar.txt` : The rewritten grammar, in its entirety (Phase I, does not include semantic actions).
- `grammar-rewrite-process.txt` : Text document showing the step-by-step changes we made when performing the grammar rewrite (Phase I).
- `compile-script.sh` : Simple script to compile all of the .java source files on Linux.
- `compile-script.bat` : Simple script to compile all of the .java source files on Windows.

Description of Compiler Steps

Scanning (Phase I): Reads in the source text character by character, and translates the characters into tokens according to a DFA. Reports and discards any erroneous tokens.

Parsing (Phase I + II): The parser receives tokens one by one from the scanner, and matches them to the grammar rules. It creates a parse tree as it matches the rules, which is traversed to perform semantic analysis and IR generation.

Symbol Table (Phase II): The symbol table is constructed during the parsing phase, according to the actions embedded in the grammar rules and the semantic stack interpretation shown in `handwritten/SemanticStack.png`.

Semantic Analysis (Phase II): The semantic analyzer walks the parse tree and ensures that the typing rules are followed.

IR Generation (Phase II): The IR generator walks the parse tree in a manner extremely similar to the semantic analyzer. It emits IR code using an unlimited number of temporary variables.

Register Allocation: Adds register allocation statements to the IR code using new IR commands `load_var` and `store_var`. Replaces all variable names in the original IR with the registers that were allocated for them. There are two different register allocation schemes that were implemented, either of which can be used during compilation (basic block allocation is the default, but command line flags can select between it and the naïve allocation scheme).

> Naïve Register Allocation: Every single statement in the IR that uses variables will load the variable values into registers from memory before the statement, then store the values from the registers back to memory after the statement.
>
> Stats for the 3 test programs:

```
test1:       instr: 1588, reads: 336, writes: 216, branches: 92, other: 944
test2:       instr: 214, reads: 36, writes: 32, branches: 7, other: 139
factorial:   instr: 245, reads: 55, writes: 28, branches: 20, other: 142
```

Basic Block Allocation: The IR stream is split into basic blocks. Each basic block identifies which variables are used within the block, their live ranges, and the number of times they are used. Variables that are used more often are given registers from the set of available registers. When all available registers (besides `$t0`, `$t1`, `$t2`) have been assigned to variables, remaining variables will be allocated registers `$t0`, `$t1`, `$t2` in a naïve way, when they are used in an instruction.

Stats for the 3 test programs:

```
test1:       instr: 1521, reads: 326, writes: 191, branches: 92, other: 912
test2:       instr: 216, reads: 40, writes: 29, branches: 7, other: 140
factorial:   instr: 233, reads: 49, writes: 28, branches: 20, other: 136
```

Because the basic blocks in the test programs were fairly short, the number of reads is not always improved much from the naïve allocation. However, since only the variables that get altered get written to, there are usually fewer writes with this register allocation scheme.

Note: A third register allocation scheme was not implemented.

Assembly Generation: This pass uses the IR generated from the register allocation pass. It then selects the correct MIPS instructions to execute the IR code. Special attention needs to be payed to whether each register contains an int or float. When operations are done on floats, they must be transferred to the MIPS FPU for the calculation, then transferred back into the result register. Array accesses also need special attention. Since all variables (ints and floats) are 4 bytes long, the index of the array must be multiplied by 4 to calculate the offset from the array's base address.

Lastly, function calls are implemented by passing arguments through special memory locations, storing `$ra` on the stack, jumping to the function, and then restoring `$ra` from the stack after the function returns. This allows for nested functions to be called. However, since the traditional method of passing arguments through special registers is not used, recursive function calls are not supported.

Compiling and Running

Open the command line and navigate to the project root directory.

- To compile on Linux, run `compile-script.sh`
- To compile on Windows, run `compile-script.bat`

Run by typing the following command:

```
java –cp src com.tiger.compiler.TigerCompiler <input-file> [-options]
```

The path to the input-file should be relative to the root directory (e.g. `test/test1.tiger`). Without indicating any options, the compiler will run silently, only outputting errors, indicating when it is completed, and writing the compiled assembly code to `<input-file->.s`. Use the following options to indicate what you would like to be printed to the console (note: the `–o` option writes all console output to an output file as well).

```
-t     : Print tokens as they are scanned.
-pt    : Print parse tree after successful parse.
-st    : Print the symbol table after successful parse.
-ir    : Print the IR code after IR generation and after register allocation.
         May not be used with the –irc flag.
-irc   : Print the IR code with comments to make it more human readable.
         May not be used with the –ir flag.
-all   : Shorthand for –t –p –st –ir. May not be used with –irc or –allc flags.
-allc  : Shorthand for –t –p –st –irc. May not be used with –ir or –all flags.
-o     : Output to file named "<input-file>.out". May not be used unless other
         flag(s) are set.
-r1    : Use naïve register allocation scheme. May not be used with –r2 flag.
-r2    : Use basic block register allocation scheme. May not be used with –r1
         flag. This scheme is the default that is used if no –r flag is given.
```

Individual Contributions

All team members contributed equally to the work.