Chris Altonji
Allen Chen
Andrew Smith

CS 4240 Phase II

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 the 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.
- `res/` : Contains .csv files of the various tables and grammar rules needed to drive the scanner, parser, and symbol table construction.
- `test/` : Contains various test programs (some with errors), as well as the output of running the compiler on them with the following flags `-allc -o`.
- `handwritten/` : hand-drawn DFA's for the scanner (Phase I), and an image of the how the semantic stack 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 found in `res/ParserProductions.csv`. It creates a parse tree as it matches the various rules, which is used in the semantic analysis and IR generation phases. It also constructs a symbol table of the user-defined IDs. If a parse error is raised, it will discard all future tokens until the next semicolon. It will then pop symbols off of the parser stack until a semicolon is found. It will then look at the next symbol in the parse stack and the next input token, and then continue parsing. If a parse error or symbol table error are raised, compilation will not continue (it does not make sense to semantically analyze a broken parse tree, or to try enforce type rules on non-declared variables/functions).

Symbol Table: The symbol table is constructed during the parsing phase, according to the semantic actions embedded in the grammar rules and the semantic stack interpretation shown in `handwritten/SemanticStack.png`. The table is stored as a Map<String, Symbol>. It's was influenced heavily by the specific rules of the Tiger language. The Symbols in the table fall within three categories, shown below:

- TypeSymbol:
    - name (String)
    - baseType (TypeSymbol)
    - isArrayOfBaseType (boolean)
    - arraySize (int, 0 if isArrayOfBaseType == false)
- VariableSymbol:
    - name (String)
    - type (TypeSymbol)
- FunctionSymbol:
    - name (String)
    - returnType (TypeSymbol)
    - parameterTypeList (List of TypeSymbol)
    - functionSymbolTable (Map<String, Symbol>)
        - params (VariableSymbols)

An important note is that there is a **single global symbol table** containing types, variables, and functions, since these can *only* be declared at a global scope. The only thing that is not declared at the global scope in Tiger is function parameters. Therefore, these are stored in the "functionSymbolTable" within their appropriate FunctionSymbol. Thus, the "functionSymbolTable" can only VariableSymbols.

Semantic Analysis pass: The semantic analyzer walks the parse tree produced by the parser, and ensures that the typing rules are followed. Some nodes have no rules that they need to enforce, while some nodes have very complex rules and many edge cases. Attributes such as type and which function scope a node is within are attached to tree nodes as they are discovered, and propagated up and down the tree as needed by parents, children, and sibling nodes. A node typically performs the following steps:

- Inherit attributes from parent.
- Assign attributes to self.
- Recursively call `analyze(…)` on children that use this node's attributes.
- Assign/update attributes of self based on attributes of children that have been analyzed.
- Recursively call `analyze(…)` on children nodes that use the intermediate attributes.
- Repeat the analyzing and assigning of intermediate attributes as needed, until all children have been analyzed.
- Assign final attributes to self based on children attributes. These attributes are now available to the parent node that called `analyze(…)` on this node.

At any step in the process, the node may check the attributes of itself, its parent, and its children, to ensure that they follow the semantic rules of Tiger. If not, it reports a semantic error. If semantic errors are raised, the compiler does not continue on to the IR generation phase.

Since different node types need different attributes that are different types and there is a only one ParseTreeNode class, we stored the attributes for each node in a Map<String, Object> where the String was the name of the attribute, and the Object was its value. When reading an attribute's value, we simply cast the Object to the type that we know the attribute to actually be. This allowed us to assign arbitrary attributes of arbitrary type to any tree node, almost as if we were using a dynamically typed language. The same trick was used for storing attributes in the IR generation pass.

IR generation pass: The IR generator walks the parse tree in a manner extremely similar to the semantic analyzer. However, instead of propagating types up and down the tree, it propagates variable and intermediate variable names. Only the nodes that actually perform actions generate code, such as a STAT_ASSIGN_OR_FUNC node, or a PRIME_TERM node.

Variables (and functions) are named identically to the variables in source code, however they are often put into temporary intermediate variables before they are used. Temporary variables begin with an underscore to ensure they do not conflict with variable names, and include a number in their name that is incremented for every new temporary variable, to ensure they do not conflict with other temporary variables. Before calling a function, arguments are placed in variables named `_funcName_arg#` (again using underscore to avoid name conflicts). When functions are declared, they assign the values in `_funcName_arg#` to `_funcName_paramName` accordingly before they are used.

Intermediate, temporary variables are used extremely abundantly, often when the values could be used directly. However, at the time of creating such temporary variables, the node that creates them does not know whether the variable is going to be assigned to something, used in further calculations, assigned to a function call argument, etc. Therefore, the compiler often creates more temporary variables than are needed, but the program remains logically correct. These unnecessary temporary variables could potentially be eliminated in further compiler passes.

Equality and inequality comparisons are calculated using branch statements that assign either 0 or 1 to the result variable, based on the direction the branch takes. Loops place labels at the beginning and end, and use branch statements to go to the end label when the loop condition evaluates to 0 (false).

Comments are included in the IR code that help a human reader know which instructions map to which lines in the source code, and which instructions are used for loop control. These comments can be turned on/off with a command line flag.

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 and indicating when it is completed. 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 the parse tree after successful parse.
-st   : Print the symbol table after successful parse.
-ir   : Print the IR code. May not be used with -irc flag.
-irc  : Print the IR code with comments to make it more human readable.
        May not be used with -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.
```

Individual Contributions

All team members contributed equally to the work.