SMACC

Thomas Grigg Kiran Patel Daniel Slocombe William Springsteen

Project Management

Our group's greatest strength was in our effective communication: we met in labs almost every day so that we could plan and work concurrently, and yet be together. Before writing any code for a milestone, we planned our designs on paper so that we had a very clear direction. These planning sessions ensured that everybody had a clear idea of what was going on. Our organisation was partly down to our use of online group messaging applications: When we were not in labs, we frequently communicated on group chats to help one another with our individual WACC goals and to schedule further lab sessions. We often split into teams of two working on separate parts of a goal. We would combine various different components of each teams code in order to achieve this goal which would usually go relatively smoothly due to our ongoing communication throughout; one team normally providing the other with a constantly updating interface to work with.

Our group made good use of Git throughout the project, our commits were regular and most messages were descriptive but concise. We made very effective use of branching in Git. A new branch was created every time we entered into a new stage of the project.
Another feature of Git that we used were tags. We used tags to mark when a significant part of the compiler had been completed, such as when the AST construction was completed, and when the compiler passed all tests on LabTS. This enabled us to quickly locate particular commits. The only criticism in our use of Git could be that we created a small number of unnecessary branches that ended up being deleted from the repository.

Design Choices

Front End

The first design decision we faced was the matter of choosing a language in which to write our compiler. We briefly toyed with the idea of using Haskell for its obvious advantages in terms of pattern-matching, but this feature was outweighed significantly by the object-oriented paradigm offered by Java once we began to develop a concrete plan of how to proceed. This decision was reinforced by the support that the department offered on the use of the ANTLR tool. Using the ANTLR tool allowed us to focus on reforming a useful and correct structure of the WACC grammar, bypassing the creation of the generic methods and patterns used to build and traverse the parse tree. This slight restructuring of the grammar allowed us to actually carry out semantic analysis on our parse tree whilst simultaneously constructing a highly refined abstract syntax tree, eliminating a second pass over the parse tree. Examples of the changes we made to facilitate semantic analysis included: differentiating between function identifiers and variable identifiers, eliminating the mutual left recursion present in the definition of `<array-type>` and sorting the precedence of binary operators. This first pass is carried out using a visitor appropriately named `WACCFirstPass` which extends

the class `WACCParserBaseVisitor` provided by ANTLR. This visitor handles all of the logic for semantic analysis, creating the symbol table and building the AST.

Handling Error-Messages with `ErrorMessageContainer`

The error message container is used to format error messages passed from the semantic analyser. The constructor takes a reference to the original source code and the list of newline positions which is built up at the beginning of the program. When an error is added to the container it is coupled with the parser node so that, as we print the error, we can also give the corresponding line of code. The `addError` function is set up similarly to `printf` where '%' is substituted from the message with the `toString`s of the objects following it.

After semantic analysis, all error messages in the `ErrorMessageContainer` are printed to `stderr`, and the code will not be further compiled. The error messages give the line and column number of each error, and are very informative, allowing the programmer to easily resolve any compilation issues.

Type-Checking Using `typeStack`

Perhaps the most fundamental part of carrying out semantic analysis was checking the validity and correspondence of types in WACC. Knowing this, we found an elegant solution to the problem: First, we created an abstract class `WACCType` which was extended by more specific type-representative classes such as `IntType` or `CharType` with methods that are used throughout the compilation process. We then create an instance of a stack in `WACCFirstPass` called `typeStack` which has an associated contract: when an expression is visited and its type is evaluated its argument types are popped from the stack and its return type is pushed onto the stack, if a type fails to match at any point, an error is passed to `ErrorMessageContainer`, and an instance of `AnyType` is pushed onto the stack. `AnyType` is a special sub-class of `WACCType` which matches with any other type - this stops an incorrect type from propagating upwards and causing unhelpful and indirect typing errors, thus allowing the semantic analysis to continue and detect the source of all semantic errors in one compilation.

Creating Scopes with `symbolTable`

Our `symbolTable` is a custom data-type consisting of hashtables (representing scopes) each of which is connected to the previous scope. Each hashtable maps variable names, represented by strings, to their corresponding `Variable` object. These individual variable objects hold the variable's type and stack offset in the symbol table. A call to the `lookupAll` function with a string identifier will return the first variable object with a matching identifier to occur in the scopes above and including the current one - if an identifier is not found, then an `IdentifierUndeclaredException` is thrown, and this causes a semantic error to be added to the `ErrorMessageContainer`. Whilst visiting the parse tree in `WACCFirstPass`, whenever a new scope is encountered, we build a special `ScopeNode` and pass it a new `symbolTable` which is marked as the `currentScope` so that the visitor can add any declared variables to it.

AST Construction and `funcTable`

The decision to construct the AST in the same pass as semantic analysis was made to pre-emptively reduce the number of passes over data-structures in our compiler. This meant handling two distinct pieces of logic in each parse tree node's visit method: Semantic analysis is always carried out first and is normally followed by the creation of the relevant `ASTNode`. If a semantic error is encountered, an invalid AST would be produced,

but this doesn't matter as the compilation process does not continue having detected errors of any kind.
An important decision we made resulted from our recognition that translation of functions could be carried out disjointly; our AST data-type is not one tree, rather it consists of a `funcTable` holding each WACC function as a separate syntax tree. We generalised this idea to handle the main body of code in a WACC program in an elegant way: the `funcTable` has a special `main` function which corresponds to the syntax tree representing this body. One potential problem that could have arisen from adopting this method is a name-clash with any user-defined function called `main`. This was easily solved: all user-defined function identifiers are implicitly pre-fixed by the string `"f_"` in `funcTable`. In `WACCFistPass` we are able to build the `funcTable` by having a `currentFunction` field whilst we do our visiting; this field is updated whenever we visit a new function declaration in the parse tree.

Prototyping

A problem we quickly encountered in relation to type-checking function calls in `WACCFirstPass` was that a function may not have been visited (and hence not added to `funcTable`). This means we have no knowledge of the function's type-signature and no way of upholding the `typeStack` contract. Our solution was to visit the immediate children of the program node and declare functions in the `funcTable` at the beginning of the first pass. The function bodies are then visited and the resultant syntax trees are mapped to the correct identifiers in the `funcTable`.

Back End

We defined a custom visitor `Translator` to walk over the syntax trees in `funcTable` and output a `LinkedList` of `ARMNode`s called `program`. The `toString` method of each `ARMNode` formats the instruction in the correct manner based on its fields, and then `program` can be written to the output stream. The decision to encode into `ARMNode`s rather than going straight to output was made to reduce the potential complexity of restructuring and rewriting any parts of assembly code, as well as to increase extensibility in terms of any optimizations on the output.

Translator

Each `ASTNode` is paired with a `translate` method which handles all the details of converting the node into a group of `ARMNode`s which are appended to `program`. Registers are utilised during the translation with the `returnReg` field, which refers to the first available register (and hence the register we want the latest evaluated expression to put its value in). There is a paradigm shift in the event that we run out of registers: the `Translator` switches to using a stack-accumulator method for evaluating expressions until registers are freed up. The class `Register` is used to make the use of registers neater, especially in the case of special-purpose registers (for which we define an enum). The `Register` class combined with appropriate use of the `OperandTwo` class in the `ARMNode` constructors results in a clean and readable syntax for adding to `program`. There are two notable special instances of `ARMNode` called `ARMFileStart` and `ARMFileEnd`. `ARMFileStart` handles the attachment of all messages used in the translated program - these messages are passed to this instance as they are translated. `ARMFileEnd` handles the outputting of predefined functions held in an instance of the `PredefinedFunctionHandler` class.

PredefinedFunctionHandler

Almost all of the logic related to predefined functions and their definitions is encapsulated within this class. A single function `addFunction` is provided to the `Translator` so that it may specify any predefined functions which become necessary as the function syntax trees are translated: `addFunction` takes a `predefinedFunction` (an enum) as an argument and simply utilises a switch statement that handles the branching to this function before delegating to a helper function to define the predefined function using `ARMNodes`. A predefined function is only added to the `HashMap` of `predefinedFunctions` if it has not already been added, thus we avoid duplicate functions in our outputted assembly code.

Product

All tests on LabTS, except the double free test cases, pass. Also, all of our custom tests pass, as well as all example programs in `wacc_examples`. Hence, our compiler satisfies the specification to a very high degree. However, a potential criticism of our semantic analyser is that there is a large amount of code and logic in `WACCFirstPass`; the logic for creating the AST is interwoven with the logic of the semantic analysis, this lack of clear separation did make the code somewhat difficult to debug. This was the trade-off we made for creating the AST whilst doing semantic analysis in a single pass.

Our compiler is set up well for future development. Lots of interfaces/abstract classes are used. If anything needs to be added to the WACC language, ANTLR can be easily updated to allow this new syntax, a new visit method for this update can be added to the semantic analyser, a new `ASTNode` can be created and a new translate method can be added to the translator. A further aid to extensibility, especially in terms of optimizations that involve restructuring the code is the flexibility of the visit method in `ASTNode`; the visit method can take multiple listeners and thus do multiple things per pass. A significant criticism of the architecture of our code is that there are points where downcasting is very hard to bypass - in these instances we are logically certain of an object's class, but rather than reform our code structure, we carry out the cast to save effort. Given more time, it is likely that we could resolve these issues by altering our design, but it is more a matter of good practise - functionality is unaffected.

Extensions (Beyond the Specification)

Function Overloading

One of the extensions implemented was function overloading. In SMACC, a function's identity no longer solely depends on its string identifier but is now also dependent on its type-signature. This means SMACC is able to determine which function a function call refers to based on the types of the parameters given when the function is called. The main change required to allow function overloading was that the `Hashtable<String, Function> funcTable` field, which mapped the function name as a `String` to the corresponding `Function` object, was changed to `Hashtable<String, HashSet<Function>> funcTable`, which maps from the function name as a `String` to a `HashSet` of `Function` objects. This extension meant that some methods related to the `FunctionTable` and `Function` classes had to be slightly modified, such as the `declare` and `lookupFunction` methods. Also, `Function` objects had to be given a `String baseId` field, which holds the overloaded user-defined function name (can be shared by multiple functions), and a `String id` field, which holds the (unique) identifier for each function. `String id` therefore corresponds to the label that is printed out in the output ARM assembly file. Explicitly, `String id` would have the following value for each function `foo` of which there are three different type-signatures: `f_foo`, `f_1_foo` and `f_2_foo`, respectively.

Abstract Syntax Tree Viewer

The AST Viewer was originally built as a way to check the integrity of our AST construction, the first plan being to build a graphical viewer similar to that of ANTLR's Grun given the `-gui` flag. However after consideration, we realised it would be far simpler and more practical to build a static viewer that produces vector images. The final AST Viewer works by walking over the abstract syntax tree using a listener pattern and produces a Graphviz source in the current directory that it then compiles with dot.
While iterating over the tree it assigns all nodes an index which is used as the variable name in the dot source. When it is declared the `toString` of the AST node is used to give a name and its type determines the shape of the resulting node.

Functions are prototyped first to ensure they are defined before they are called. We have two display modes: structured and unstructured. Structured lays functions out horizontally with no overlapping or arcs between functions; unstructured places arcs on call nodes causing the Graphviz algorithm to position functions within others. The latter allows you to see the logical structure of the program yet the former better relates to the generating WACC source code.

The viewer can be accessed with the `--graph` flag followed by a filename to output to. The format can be changed with the `--format` flag, it supports Portable Network Graphics (PNG), Scalable Vector Graphics (SVG) and Postscript (PS). We default to a structured layout but unstructured graphs can be created by setting `--unstructured-graph`. It is also possible with the `--graph-no-compile` flag to leave the dot source uncompiled, full documentation and usage can be found by running the program with `--help`.

Compile-time Constant Evaluation

Wherever possible, arithmetic expressions are evaluated at compile-time. This means that all numerical constant expressions are reduced to a single constant literal before assembly code is generated. This is implemented within the first pass so that the function syntax trees hold no complicated constant expressions.

Immediate Operation Handling

Rather than loading all literal values into a register and performing operations using registers as arguments, SMACC attempts to save registers for the use of identifiers by instead utilising the immediate forms of the ARM instructions corresponding to each operation (where applicable). This, combined with compile-time constant evaluation allows us to significantly reduce the number `LDR` and `STR` operations in a relatively simple manner.

Future Prospects

Given more time, we would like to implement some form of efficient register allocation, such as the use of graph colouring. This would optimise code generation, specifically on very large programs such as `Tic-tac-toe`, where `Translator` is forced to switch to the stack-accumulator paradigm. We did have plans to implement this, and we even got as far as carrying out an effective live-range analysis on our function syntax trees. However, we didn't have time to complete the new feature and so it was regrettably dropped from the project.