CIS*4650 - Compilers Milestone 3

Boris Skurikhin, Brayden Klemens

April 2022

M3 Documentation

Techniques:

- We used an array of helper “emit” functions to make our code more readable.
- Used switch statements when possible to improve code readability.
- For array pointers (pass by reference), we created a boolean flag stored in the ArrayDeclaration object. If an int array is created from function parameters, we set the “passAsAddress” flag to true. This flag decides how load the array upon access (read/write to some index). This technique enabled us to pass arrays by reference while maintaining a single real address.
- Used temporary variables, and emitSkip to store return locations when working with things like conditions and loops.
- Used global variables to keep track of things like global offset when working with declarations of global variables.
- We used a queue to store all global variable declarations, and print them out to the file only if the main function is found.

Design Process:

- The design process began with modifying the visitor pattern interface. The two modifications made included adding an offset parameter, is address boolean, and changing the function return types from void to int. The latter change helped us

track offset more easily for things like variable and array declarations (without having to use global variables). We did not want this change to affect the code for the other two milestones so we simply created a new class called AbsynVisitorM3. We then declared some global variables representing various registers like ac, pc, fp, gp and ac1. We also created the symbol table. When writing the code generation logic, we broke our development process into small milestones - the first one being allocation of an array in memory along with other variables. The order in which we added code generation logic is as follows: variable declarations, function declaration, compound statements, variable expressions, assign expressions, math expressions, return expressions, call expressions. Every now and then, we would compile some test code and step through its instructions on the emulator ensuring that it works as expected. Lastly, we modified our code by enabling passing arrays by reference. As mentioned above in “techniques”, we did this by adding a special boolean property to the arraydeclaration class - and checking this property when accessing the array.

Lessons Gained in the implementation process:

- One of the key lessons we gained during M3 was the importance of checking to make sure a particular function of the generator worked before adding more features. By making assumptions that our implementation worked, it made it very difficult to debug when trying to build more complex tm-files. Sometimes we later found side cases would give us null pointer exception and result in an empty tm-file.

- A frustration that occurred was accidentally running our code generator a few times without the -c flag. Due to lack of any warning messages (that the flag was not specified), we were led to believe that whatever output file we had open was the version we just compiled. This led to about 40 minutes of confusion due to the compiled program not responding to changes made in its source code. We prevented this from happening in the future by temporarily adding a gentle reminder when a flag is not passed (like -c).
- Remembering to do the recursive descent, otherwise some visit functions would not be found.
- One thing we were stuck on for a while was referencing global variables. A realization was made that if the variable scope is 0, we should be using the global pointer (gp). If the variable has scope higher than 0, we should be using the frame pointer (fp). Although this information was in the slides, we missed it and ended up struggling for a while. The key lesson here is to fully familiarize ourselves with all the registers next time before jumping into the programming process.
- Adding a boolean flag to array declaration, stored in the symbol table, made it easier to check if it was passed by reference or not.

Assumptions:

- We assume that the programs provided follow the c-minus specification. Otherwise the compiler will not be able to generate working assembly code.

- We assume the runtime environment has at least 1024 slots of memory. We're making this assumption because we have only tested our programs with this memory setting in the tiny machine.

Limitations:

- Our compiler is not able to handle run time exceptions such as caching index out of range
- The only RE we catch is missing main function

Possible Improvements:

- Add in the functionality to handle more run time exceptions these exceptions can include things such as:
    - Index out of range
    - Divide by zero (although this is handled by the TM simulator)
    - Integer value overflows
    - Integer value underflows
    - Infinite recursion detection
    - Dead code detection
    - Negative numbers
    - Tracking unused variables

RunTime Errors Handled:

- If the main method (void or integer) is not found the program will result in an immediate HALT and a run time error.

Contributions :

Boris:

- Code Generator helper functions, Code Generator visit functions, supporting logic flow in Main.java method.

Brayden:

- Code Generator helper functions, Code Generator visit functions.

Compilers Project Overview

Major Changes between milestones:

- For the most part, building the compiler went fairly smooth. We did not have to make any major changes to previous milestones, however we were able to make minor additions to some code which helped us in the implementation of M3. A good example of this is overloaded new constructors in our absyn class structure, to allow for address and offset tracking. We also added the offset property to our ANode class for milestone3.

Major Issues Encountered:

Milestone 1:

- Had a difficult time catching global scope syntax errors, possibly due to the way the grammar is specified. A generic syntax error is shown for this.
- Having to define precedence

Milestone 2:

- Keeping track of global vs local scope declarations. The code for this part was messy and all over the place resulting in us using global variables.

Milestone 3:

- The most difficult part was passing arrays by reference.

- Did not have enough time to add runtime errors

Error Handling:

Milestone 1: Error Recovery

- Empty files are not allowed.
- Invalid variable declarations such as: improper names, array sizes, and types.
- Invalid parameters in functions such as declaring a value.
- Invalid statements (if, else, loop, return). This occurs when the expressions inside of the statements are invalid. These include things like improper variable names and bad comparisons.
- Function calls cannot have invalid function call parameters.
- When assigning array values, the array cannot have an invalid index.

Milestone 2: Semantic Errors

- Redefining variables / array variables at the same scope
- Redefining a function a the top level scope
- Int return type functions that do not return a value
- Void function attempts to return a value
- Variables not being declared before being used
- Array variables not being declared before being used
- Functions not being declared before being called
- Calling void functions as part of integer expressions (ex: k = foo(); foo is void)
- Assigning functions to improper types (variable not declared or check type)
- Function arguments mismatches (‘array expected’ or ‘int expected’)
- Too many function arguments passed to a function call
- Insufficient function arguments passed to a function call

Milestone 3: Run Time Errors

- Halt immediately if no main function is found

Attempted Repair of Detected Errors:

- For error recovery during milestone 1, we recovered from errors by inserting special ERROR tokens if a syntax error was encountered.
- For error recovery during milestone 2, we caught errors, reported, then continued execution.

Source Language Issues

- void variables detected and reported at stage 1 of compilation