Compilers Project Checkpoint 3

CIS*4650 - Compilers

Benjamin Bliss (1146484)

Uladzislau Kaparykha, (1096425)

Construction of the Compiler

In this project we are developing a compiler for the language C-, a simplified version of C. The compiler was built over 3 major checkpoints in which progress was made over major increments. In the first checkpoint we implemented lexical analysis using JFlex and constructed a parser with CUP to generate an abstract syntax tree (AST) while ensuring related errors were recovered from so parsing could continue. In the second checkpoint we built a symbol table alongside type checking validating the use of all variables, functions and comparisons are all correct. Finally, in the final checkpoint, we implemented generating assembly code for valid C- programs that can be executed on the TM simulator. With all these components completed, our compiler is now in a completed, fully functional state.

The first component in the compiler was the scanner, made with JFlex which was implemented to tokenize the C- source code and lexically analyze it. Special keywords and special symbols such as such as if, int, return, void, +, -, *, /, etc. all get matched. Whitespace gets ignored and exists to separate tokens. Comments are matched with the pattern /* comment */ and //commend which get ignored by the lexer. Unrecognized symbols are identified and reported here.

The next step we took was to build an abstract syntax tree (AST). This step was relatively straight forward as we used the Class based AST for C- structure as provided in the class slides as a blueprint for our AST. This AST is used to formulate an intermediate representation of the source code that the parser builds during the parse phase. Ultimately when the program is run, the AST gets printed in a human readable format when the program is run for the user to verify and debug.

Parsing with CUP was the largest step in the first milestone. The objective of the parser is to validate the syntax of the inputted C code based on a defined grammar. The main core of the cm.cup file is the grammar section which is a list of context free grammar rules for the C- language. The declared rules had to go over a few revisions, first the bare rules which only contain the grammar, next embedded code was added to the rules to produce the syntax tree, next modified existing rules to set precedence and nonassociativity, and finally error handling which is described more in the next section.

For the second milestone, we began by building the symbol table which we implemented as a stack of hashmaps. The hashmaps used strings as the key, holding the identifier and a new class Sym which held a string for the name, the AST declaration and an int for scope. We chose a stack of hashmaps as it streamline the process of entering and exiting scopes in a source file. Such that when the program starts, the first hashmap gets pushed on representing global declarations and each time a local scope is entered a new hashmap is pushed to hold variables declared in that scope. Whenever a scope is left, the top hashmap is popped from the stack and its contents are printed to show all declarations in that scope. Next, the foundation of the SematicAnalyzer was built, which is structurally similar to ShowTreeVisitor of the last assignment, traversing the tree in post order, inserting symbols into the symbol table and reporting errors if the symbols are already contained in the current scope.

As for type checking, all of the individual checking was done pos-order within visit() nodes of SemanticAnalyzer. The contents of the node would be recursively visited, and then type checking would be performed. Type checking mainly checks if the provided contents is of ArrayDec type, if yes, raise error. Afterwards, where applicable, it checks if dtype (predefined in Exp class) is of a correct type, given the context (eg. must be either int or bool for a test condition of an if-statement, etc.) In terms of CallExp(), we do a hashmap lookup to see if it’s a valid function, do appropriate checking to verify provided ars match the datatype of the provided function; and then set dtype of CallExp to return value of predefined function s.t. callee can use dtype to do appropriate checks. In terms of ReturnExp, we keep track of the function name's current scope as a class-wide variable, and then use that to do a hashmap lookup to validate the return type.

Finally for the third milestone, we implemented assembly code generation. This task was very dense as understanding the TM Simulator architecture and its assembly code was challenging. We began by Refactoring the syntax tree and visitor interface. This required updating and changing the existing AST, first adding a flag argument to all the accept and visit functions and next updating a few of absyn subclasses. A pointer to the declaration of any Var or function call was added such that we would know where to reference when using variables/functions. The declarations of variables were updated to store whether they were global or local and their offset so we could keep track of where in the stack they were.

Next, we started writing emit calls to output assembly instructions in each of the AST visit functions starting with the prelude which sets up the program, defining the io functions and the finale which wraps the AST and all the assembly code it generates. Each visit function had to be handled differently, the first one completed was FunctionDec as it is required for main and thus needed to get anything started. The basic structure of it is to skip and save the current line then call the accept function's parameters and body to process everything inside, then backpatch in the skipped line to jump just past the function. We would continue this process, to the next relevant absyn visit and update it to generate the relevant assembly code. Often only enough to get it working and then revisited once we better understood how different absyn structures relied upon each other.

Error handling in checkpoint 3 was minimal, consisting mainly of array indexing, and making sure the index was within the bounds of the allocated array. If not, the program terminates early. The main issue that arose from translating AST into assembly code was all related to the way FunctionDec was handled, which was already explained. However, when writing CodeGenerator and having to debug it, referring to the generated AST to know what to expect helped a great deal.

What we learned

The main lesson that was gained during this process was how to handle the function calls when the function was not defined yet, but a prototype was given. Our solution was to redefine how AST is generated where if an FunctionDec with an existing name is found, and the body is empty, we update the existing one's body. Additionally during code generation, when a bodiless function is called we keep track of where that is. As such, if there is a call to a function that doesn't have a location within the instruction set yet, we can add a back patch location to the list kept within FunctionDec; and when the function is defined, for every element in that list, backpatch and jump to the correct location.

Another lesson to be learned was how to handle array indexing, and how those arrays relate to where they have been defined: whether passed in as part of the function call, or defined locally. To approach that, we determined that if the array was passed in, its reference to ArrayDec will be pointing to the definition within function parameters, which will have size 0. Since size 0 is not allowed (as per semantic analysis), that means the declaration the array is referring to must be passed in. Otherwise, the declaration is local, and must be handled accordingly.

Possible Improvements

The main improvements that could be made for our compiler center around optimization. Currently, the implementation decrements the offset by 1 for nearly every visit (excluding calls, that is handled separately). And every AST node that has any type of return value will be stored in the provided offset. And then the parent of the AST node retrieves the value stored in that offset and works with it accordingly. However, there are 2 issues with this approach, and thus can be further optimised: First, not every value needs to be stored in the offset. Currently, a number of numbers not only store the value in the offset, but also store the value in AC. And as such, the calling parent can also retrieve the value from AC instead and work with it accordingly. In terms of optimization, a compiler can be reworked to instead work with AC values directly, and store them if needed, as opposed to “store everything, and sort it out later”. Second, not every visit node needs a decremented offset to store a variable. If the tree sequence is VarExp>SimpleVar>{retrieve value}; offset can stay consistently 1, as opposed to decrementing for every visit. This will allow for better memory usage, and less likely to run into stack overflow (or equivalent).

Work Breakdown

- Ben: visitor class, absyn modification, control structures code generation
- Vlad: main, functions and calls, array handling code generation, further modifications to absyn

Assumptions and limitations

- Array declarations only check if the size of the array is too small (<1). It does not check for arrays too big, and thus will be a runtime error.
- Warnings are not distinguished from errors, they are all reported in the same way (eg. function returns void, but has `return int`)
- All C like comments are allowed eg: “//”
- The operators == and != only work with bool data types
- If a function is not void it must have a return statement with a matching type
- All function prototypes must be implemented.
- There is no error handling for mismatched braces
- The compiler will not proceed to the next stage if an error is encountered (eg: will not continue to Semantic analysis if a syntax error exists)