COMP3290 Report Brandon Zahn c3186200

Changes to the CD18 grammar

I have made some changes to the grammar to make it a LL(1) grammar:

- ***Left recursion:*** Rules that contained left recursion were rewritten to eliminate the most left productions in rules being stuck in an infinite loop. This was solved by calling a terminal function that is part of some production rule **A** of the set of FIRST(**A**).
- ***Left factoring:*** By reorganizing the parser, I have eliminated some ambiguities to parse as LL(1) grammar

Semantic Analysis Overview

All of the semantic checks listed in the assignment sheet have been successfully implemented:

- ***<id> names (arrays and variables) must be declared before they are used:*** Here, each time the parser comes across a NSIMV or NARRV node, I check if the name exists in the symbol table for the given scope. If it does not exist in the symbol table for the scope where the variable was encountered, then it has not been declared.
- ***Array size – must be known at compile time:*** Here constant folding is performed to resolve the expression field in the [] part to some integer constant literal.
- ***Strong typing exists for real variables, real arrays, boolean expressions, and arithmetic operations:*** Here, for each expression node, I traverse down until I find a leaf node, and add the type of this leaf to a set of String. I do this for all nodes under the root expression node, WITH THE EXEPTION of nodes that belong to a NARRV index expression node, a function call's list of expression nodes for parameters and the NARRV member id node. Once all leaves have been traversed underneath the root expression node, I check the size of the set. If the size of the set is greater than one, it must mean that there are more than one type of variable present in the expression, thus meaning it is not a strongly typed expression.
- ***Valid assignment operations:*** When an assignment node is encountered, the type of the left hand side variable is compared to the type that the right hand side expression resolves to. If these two types are not the same, it is an invalid assignment operation.
- ***Actual parameters in a procedure or function call must match the type of their respective formal parameter in the procedure definition:*** When a function is defined, a special function symbol is created that contains, as attributes of the symbol, a list of parameters and their respective types. So when a function call is encountered, each of the expressions in the function call's parameter fields is compared to this list in the function definition symbol. If a parameter type in the function call does not match with the respective function definition parameter type, it must be an invalid function call.
- ***The number of actual parameters in a procedure call must be equal to the number of formal parameters in the procedure definition:*** Since the list of attributes for all functions is kept in the function definition symbol in the symbol table, determining whether a function call is using the correct number of parameters, I compare the number of expression nodes in the expression list of the function call to the size of the list in the function definition symbol. If they are not the same value, then it must be an invalid function call.

- ***A function must have at least one return statement***: To check if a function has at least one return function, I add an entry to the symbol table for the function scope. A simple Boolean variable to say that at least one return statement has been encountered while the parser has been in the function scope. So each time the parser comes across a "func" keyword, the global scope variable is updated to the function's name. If the Boolean is false when all the statements for a function body have been processed, it means that no return function was present. So it is an error.
- ***Additionally, the returned expression type of a function is compared to the formal return type in the function definition:*** Once the body of statements of a function are processed, the returned expression type is checked if it is valid.
- ***<id> names must be unique at their particular block level (scoping):*** As mentioned above, there is a global variable String in the parser that defines the scope that the parser currently is in. Starting with "@globals", arrays defined in the globals section is visible to all sub-scopes in the program. Then, each time the parser comes across a "func" keyword, the global scope variable is updated to the function's name, and when it enters the main function, the scope is updated to "@main". The "@" is used to distinguish compiler keywords that I have created from possible conflicts with tokens from the source code.

I have not implemented the constant array type semantic checks using the node type NARRC.

Code Generation Overview

All of the following aspects of code generation have been successfully implemented:

- ***Function calls:*** All aspects of function calls have been implemented with the exception that a function is defined before it is called in the source code. Functions that have already been defined in the order of the source code can be called in following functions, allowing for some limited recursive calling between functions. It is also possible to call a function within another function's parameter field (like `func1(funcReturnsInt(0))`), assuming that the function call is semantically correct with the formal function definitions.
- ***Code optimization:*** There is some optimization with the loading of values in the main locals field. Instead of using STEP instructions, I count the number of variables that are in the list, and then use ALLOC to allocate the required amount of words on the stack. Also, with integer literals that are less than 256, I use the LB command and for integer literals with a value more than 255 and less than 65,536, I call the LH instruction.
- ***Constants:*** All the constant fields types work for String literals, real and integer literals.
- ***Function environments:*** The use of the base pointer 2 was required to successfully implement functions. As mentioned above, functions can be called within functions.
- ***Arrays:*** Arrays can be used in all aspects. An array copy assignment is also possible, such as the following assignment statement (`arr1 = arr2`) where the arr1 will then simply take a new reference to arr2 array descriptor. This assignment does not work in the case of assigning arr1 to a new reference, and then using arr1 in following function calls. It will still refer to its original value. This is because the symbol in the function instructions is not updated, and I'm not sure how to handle this case. So in other words, it only works in main and the current scope.
- ***Repeat stats:*** Works, in main and in function environments.
- ***For stat:*** Works, in main and in function environments.

- ***Ifth stat:*** Works, in main and in function environments.
- ***Ifthe stat:*** Works, in main and in function environments.
- ***Input:*** Works, in main and in function environments.
- ***Print:*** Works, in main and in function environments.
- ***Printline:*** Works, in main and in function environments.
- ***Return stat:*** Works, in main and in function environments.
- ***Address Access:*** Works, in main and in function environments.
- ***Asgn stats:*** Works, in main and in function environments.
- ***Bool stats:*** Works, in main and in function environments.
- ***Logop stats:*** Works, in main and in function environments.
- ***Relops:*** Works, in main and in function environments.
- ***Expr:*** Works, in main and in function environments.
- ***Terms:*** Works, in main and in function environments.
- ***Facts:*** Works, in main and in function environments.
- ***Exponents:*** Works, in main and in function environments.
- ***Intlits:*** Works, in main and in function environments.
- ***Floatlits:*** Works, in main and in function environments.
- ***Boollits:*** Works, in main and in function environments.
- ***arrayAccess:*** Using the array descriptor and the INDEX instruction, it is possible to access array element values.
- ***ArrayAddressAccess:*** Using the same method above, but calling the LOAD VALUE @ ADDR instruction to access element addresses.

Extensions to CD18 and the SM Environment

Proper use of the power operator as described in the SM ARCHITECTURE with real variables. Also, the std in and out could be put into some box in the GUI to simulate a terminal a bit better.

Example program: Polygon Calculator

Attached is a polygon calculator program called `Polygon.cd`. It is there to help show what the compiler is capable of. It is in the folder called Polygon and contains some std input file as well.