CIS 4650

Project - Final Delivery

Jorge Luiz Andrade
0906139

Symbol Table

The symbol tables was done for functions and variables types. Typedefs, structs and arrays are not being treated.

As for checkpoint 2, only one symbol table was being used. It caused some problems regarding functions and scoping, so a second symbol table was created just to store functions names and parameters.

The table is construct as a array(for storing the hashed index) of lists(for collision within the hash) of stacks(for storing different scopes).

When a symbol is to be stored, a hash function calculates it's index based on it's name and stores it at the end of the bucket list. Although the stack's functions are coded, the scope's treatment is not being done yet.

Type Checking

The type checking is working for both variables, function return, function parameters and constant values. It was not implemented for typedef,struct and arrays.

The check is done by comparing the type of the node being available with the type stored in the symbol table. For constants, the type is checked directly based on the constant type.

Error Reporting

Semantic error is being reported in cases of: Incompatible types, undeclared variable and function, incorrect number and types of function parameters.

The error is propagated upwards in the tree until the analysis can be followed correctly. For this reason, only one error is reported for each chain of statements, for example: if a is a variable of type int, and b is na undeclared variable, **a = 1.5 + 2.5 + b** reportes only the first error on the tree (undeclared variable on this case).

Scoping

The functions for entering and exiting scopes are implemented. When reaching a new scope a special symbol is pushed in every stack of symbols, for example:

a : int -> DELIMITER -> float

Which means that at the current scope *a* is a float.

When leaving a scope, every stack is popped so that the last scope is reached.

Although the functions for entering and exiting a scope are implemented they were commented out as they were causing some issues when translating functions.

Supported Elements

The translation from Cflat to Assembly is fully implemented for the following elements:

- Operations on integers and chars(variables and constants), both arithmetics and boolean;
- Conditionals(if and if-else)
- Loops(while and for)

These elements weren't implemented:

- Arrays, structs and typedefs
- Functions(including scoping)
- Operations on floats
- Increment and decrement(++ and --)

Intermediate Representation

The intermediate representation was done using quadruples, as seen both in class and at the Dragon Book. It uses 4 fields: operation, argument 1, argument 2 and result. A field for a label was also included.

The translation to the intermediate representation is done alongside with type checking. The syntax tree is transversed and each node is evaluated. When needed, a node sends back information to its parent, such as the name of the result or errors.

For operations like *a + b + c* a *temp* is generated for the intermediate steps. A static counter is used to keep track of the current name. To avoid conflicts with user defined names, the temps are named as *0tmpx* (where x is the current index), since Cflat doensn't allow variables names starting with a number, which is checked before, at syntax level.

Labels are created as needed and a static variable keeps track of the next label name. Each label is named as *Lx*(where x is the current index)

MIPS generation

MIPS code is generated directly from each line of the intermediate representation and using the symbol table. For each operation it is check basically: the type of the operands and if they are constants or variables. As the MIPS doensn't have immediate operations for floats, it was preferred not to implement them and focus on finishing conditionals and loops.

For each operand, it is checked if it is a constant or a variable. As all operands are saved as strings in the intermediate representation, this checking is done by iterating over the string and testing if each character is a number, in which case, the operand is a constant.

If he operand is a constant, the value is extracted directly from the intermediate representation(using C *strtol* function). If it is a variable, the name is searched on the symbol table and it's location is returned.