P2 report

Ruoyu Wu

Implementing SymbolTable

constructor and destructor

In the constructor, we create the root ScopeTable and dummy identifiers following instruction. The createIdentifier() adds the dummy identifiers to the ScopeTable. In the destructor, we just delete the root ScopeTable.

isDeclaredInScope

We use the root ScopeTable's `searchInScope()` to check.

createIdentifier

We first check if the name is already taken using `searchInScope()`. If it is, we return the identifier retrieved by `getIdentifier()`. Otherwise, we add create a new identifier, and add it to the scope table.

getIdentifier

We use the `search` function in scopeTable to get the identifier.

enterScope

We create a new scopeTable with the current one as the parent, and return it.

exitScope

We return the current scope's parent scope.

Implementing ScopeTable

constructor and destructor

In the constructor, if the scopeTable has parent, we add the newly-created scopeTable to its parent's children. In the destructor, we free all of its identifiers and all of its children scopeTable.

addIdentifier

We add the identifier to scopeTable's mSymbols.

searchInScope

We search the name in mSymbols, and return the identifier if found.

search

We first try to find the identifier in the current scope, and return it if found. If not found, we call its parent's scopeTable to search for it.

Integrating the Symbol Table

Parser::parseCompoundStmt

When parsing a compound statement, if it is not a function, we enter and exit the new scope accordingly.

Parser::getVariable

We retrieve the variable from the symbol table and return. If it is not found, we report semantic error and return the dummy identifier.

Parser::parseDecl

We insert a snippet of code to check the identifier to be declared is not declared before. If it is, we report a semantic error.

Implementing type conversion

charToInt

If the `expr` is of int type already, we just return it. IF the `expr` is of char type, there are three cases listed below: (1) if it is `ASTConstantExpr`, we just change the internal type to int and return; (2) if it is `IntToChar`, as the instructions, we just grab the child and return; (3) otherwise, we create an `ASTToIntExpr` node with `expr` as its child, and return.

intToChar

This implementation is opposite of `charToInt` described above, so it is omitted here.

Integrating charToInt

We just add `charToInt` just before returning the expression.

Integrating intToChar

parseAssignStmt

We first get the type of `ident` and `expr`. If `identType` is of array type, then we report an error since `expr` cannot be of array type at that program point. Then, if the `identType` not equals to `exprType`, we check if it is from int to char, which is the only valid conversion. If it is, we convert the `expr` with `intToChar`. Otherwise, we report semantic error.

parseDecl

This implementation is similar to `parseAssignStmt`, so it is omitted here.

parseReturnStmt

We check if the type of return expr equals to the type of function return type – we only allow int to char implicit conversion.

Implementing type checking

We just add the type checking to four `finalizeOp`, and add checks in all the functions that invoke thse `finalizeOp`.

Return statements

parseCompoundStmt

If the parsing compoundStmt is a function, we check if its last statement is return. If not and the return type is void, we add a return statement. If the return type is not void, we report semantic error.

About report line and column number

In order to align with the reported column and line number in the test cases, we override the column and line number in some `reportSemanticError`. In general, we use the column number before consuming any token.