CM4016 Languages and Compilers

Coursework Implementation Report – Part II

Tsonyo Vasilev

1. Semantic Analyser Rules

- The skip command ‘ ’ has no effect when executed (completed)
- The assign command is executed by evaluation the expression to a value then the variable is updated to this value (completed)
- The procedure calling command is executed by evaluating the parameter list to a list of parameters the procedure identified and then is called with that list (completed)
- For sequential command: single-command1 is evaluated first (completed)
- Bracketed command is executed by simply executing the command (completed)
- The block command is executed by handling the declarations first then the command executed (completed)
- The if command is executed by evaluating the condition and executing first expression if it’s true, otherwise the second expression (completed)
- The while command is executed by evaluating the condition and if it’s true, execute the command and repeat until condition is evaluated to false. (completed)
-

1.1. Semantic Analyser Implementation

- Create and maintain the symbol table (completed)
- Check declarations and variable use (completed)
- Perform type checking on the source program (completed)
- Enforce program scope (completed)

2. Semantic Analyser

Semantic Analyser is the third part of the compiler. It takes the abstract syntax tree produced by the Parser and produces an annotated abstract syntax tree. At first the syntax tree provided by the parser is just a representation of the source program. That representation is not semantically correct which means that it might have errors regarding variable declaration, types and usage problems which the Parser has not identified. This is where the semantic analyser comes in play. The job of the analyser is to run through the syntax tree provided by the parser and identify any errors regarding variable declaration, type, usage and scope.

Another functionality that the analyser implements is the symbol table. The symbol table holds a representation of all variables and functions as well as their type and level scope which is how the analyser can identify whether a variable is being used out of scope or a function is returning wrong type. Once the error check has finished and there have been no errors identified then the analyser produces what is called an annotated abstract syntax tree, which is an abstract syntax tree that holds the internal representation of the source program and is semantically correct. That syntax tree is then passed to the code generator for further checks.

2.1. Create and maintain the symbol table

The symbol table is created and maintained in a class called IdentificationTable. The representation of the table is a List which consists of Dictionaries that consist of a String key and Declaration values. The declaration values are the abstract syntax representation of the variables that are stored in the table. The OpenScope( ) and CloseScope( ) methods insert or remove new, or existing dictionaries in the list which represent a new level of scope whenever a nested code block is encountered or closed. Those methods are needed to enforce scope on the source program. The Enter( ) method is used to insert new variables into the symbol table whenever they have been defined. As stated above the details of each variable which include the name and abstract syntax declaration are inserted at the current scope level. And finally the Retrieve( ) method retrieves a desired variable from the symbol table. This method is used for checking whether the desired variable exists in the current scope level or whether the variable has even been declared, as well as the type of the variable.

2.2. Check declarations and variable use

The declarations and variable use checking is done using the symbol table through the Enter( ) and Retrieve( ) methods. When variables are declared, the symbol table checks whether there is already a duplicate existing in the table. If there is no duplicate the declaration is inserted in the table. Whenever a variable is being called in the code, the symbol table uses the Retriever( ) method to check whether the desired variable exists and returns all the information about it.

2.3. Perform type checking on the source program

Type checking once again is being done using the symbol table. Whenever a variable is being used in the code, the symbol tables retrieves its type and value. The type and value is then used to execute whatever command the variable is being used in. The reason why type checking is needed is, because it prevents illegal operations to be used on two variables of different types.

2.4. Enforce program scope

Enforcing program scope is also executed using the symbol tables. This is where the helper methods OpenScope( ) and CloseScope( ) are needed. Since those methods define dictionaries that hold representation off different scope levels, the source program can now request variables and check whether the requested variables have been defined at the current scope level. This ensures that variables defined in lower scope levels cannot be called in upper scope levels.

2.5. Semantic rules and the checker class

Semantic rules define in what order the syntax tree should be built before it can fully become an annotated abstract syntax tree. This is done using what is called the Visitor Pattern. The visitor pattern is used to run through the code and run a full semantic check without executing the code and at the same time building the annotated abstract syntax tree. Visitor pattern also handles the error reporting and updating the symbol table.

3. Checker – Commands

3.1. Visit Assign Command

This method calls the visit method on the Vname extracted from the syntax tree. The visit method is then called on the expression from the syntax tree and finally some error checking and reporting in the case of errors is done.

3.2. Visit Call Command

Calls the visit method on the identifier and then converts the stored variable into an IProcedureDeclaration. If the variable is not null, the visit method is called onto the actual parameters. Otherwise an error is reported.

3.3. Visit If Command

First the visit method is called onto the expression. The expression is then checked for errors and if there aren't any, the visit method is first called on the true command and then on the false command.

3.4. Visit Let Command

The visit let command works a bit different from the previous methods, since it defines a new nested block, so in this case the symbol table has to be used. First, the OpenScope( ) method is used to show that a nested code block has been encountered. The visit method is then called on the declaration and the command, and finally the CloseScope( ) method is used to declare the end of the nested code block.

3.5. Visit Sequantial Command

The visit method is first called on the first command and then on the second to make sure the first command is executed with priority.

3.6. Visit While Command

The visit method is called on the expression. Then errors are check and reported if any and finally the visit method is called on the command

4. Checker - Declarations

4.1. Visit Const Declaration

The visit method is called on the expression and then the newly declared const variable is submitted in the symbol table. If the variable has already been declared an error is reported.

4.2. Visit Var and Type Declarations

For both these declarations, the visit method is called on the variable type and then the variables are submitted to the symbol table. If variables have already been declared, errors are reported.

4.3. Visit Sequential Declaration

In this case the visit method is called on the first declaration and then on the second.

5. Checker – Expressions

5.1. Visit Binary Expression

The visit method is called first on the left expression and then on the right. The visit method is then used on the operator and it is stored in a variable. This variable's type is converted to BinaryOperatorDeclaration and it is checking if it has a value of null. Depending on the variable value, it is checked for incompatibility.

5.2. Visit Call Expression

The visit method is called on the identifier and the outcome is stored in a variable. The variable is then converted to a type of IFunctionDeclaration. If the variable value is not null, the visit method is called on the actual parameters and the abstract syntax three type is returned. Otherwise error is displayed.

5.3. Visit If Expression

The visit method is called on the first expression to evaluate the condition. Then the true expression is visited and after that the false expression. Finally, the initial condition type is returned.

5.4. Visit Let Expression

Since let expression indicate the start of a nested code block, that means that the OpenScope( ) symbol table method needs to be called to enter a new scope level. The visit method is called on the declaration and then on the expression. Finally, the CloseScope( ) method is called to indicate the end of the nested code block and the type of the syntax tree part is returned.

5.5. Visit Unary Expression

The visit method is first called on the expression and then the operator visit method call is stored in a variable called binding. The operator type is then enforced to change into a type of UnaryOperatorDeclaration and the result is stored in a different variable called ubinding. If the value of ubinding ends up not being null, an error is displayed and the ast type is set to the Type Denoter stored in the ubinding variable and the result is returned. Otherwise, a different error is displayed and the ast type is set to the error type.

5.6. Visit Vname Expression

The visit method is first called on the Vname and the ast type is set to the Vname. The result is returned.

5.7. Visit Character/Integer Expression

Since these methods are dealing with terminals, all that needs to be done is to set the ast type to the terminal type depending on the method, and the result is then returned.

6. Checker – Parameters

6.1. Visit Const Actual Parameter

The visit method is called on the expression. Then the FormalParameter is forced to change its type to ConstFormalParameter and the result is stored in a variable called param. An error is returned depending on whether the param variable value is null or not null.

6.2. Visit Var Actual Parameter

The visit method is first called on the Vname. The actual parameter is then checked if it is a variable. If it is not a variable, an error is returned. If the formal parameter is of type VarFormalParameter, it is stored in a variable called parameter and an error is displayed. In all other cases, a var parameter is not expected there, therefore an error is returned.

6.3. Visit Multiple Actual Parameter Sequence

First, the formal parameter sequence is set to a type of MultipleFormalParameterSequence. If the result is not null, the visit method is first called on the actual parameter and then on the actual parameter list. Otherwise an error is returned.

6.4. Visit Single Actual Parameter Sequence

The formal parameter sequence is set to a type SingleFormalParameterSequence. If the result is not null, the visit method is called on the actual parameter. Otherwise an error is returned.

7. Checker – Terminals

7.1. Visit Character/Integer Literal

Because those two methods are dealing with simple terminals, only the terminal type is returned.

7.2. Visit Identifier/Operator

In both methods the desired terminal is being retrieved from the symbol table. If the retrieving is successful, the declaration is set to the returned value from the symbol table.

7.3. Checker - TypeDenoters

For VisitSimpleTypeDenoter( ), The visit method is called on the identifier and the result is changed to a type of TypeDeclaration. If the outcome is not null, the type is returned. Otherwise an error is returned. In all other methods, simply the denote type is returned.

7.4. Checker – Vnames

The visit method is called on the identifier. If the result is not null, the type of the identifier is returned. Otherwise an error is returned.

8. Code Generator

Code generator is the final part of the compiler. It takes the annotated abstract syntax tree, which is the output from the Semantic Analyser and turns the syntax tree instructions into machine code instructions which can then be executed by the interpreter. The code generator uses two main classes in the translating process. The Encoder visits the annotated abstract syntax tree nodes, and then the Emitter writes out the machine code based on the templates, which ends up being an instruction set for the target machine.

8.1. Entity Class – Known Address

8.2. Encode Assign

The emitter emits a STORE instruction which pops an object from the stack and stores the address provided, using the frame size, display register and address displacement.

8.3. Encode Fetch

The emitter emits a LOAD instruction which loads an object from the provided address and pushes in on the top of the stack, using the frame size, display register and address displacement.

8.4. Encode Fetch Address

The emitter emits a LOADI instruction which pops a data address from the stack, fetches an object from that address and pushes it onto the stack, using the frame size, display register and address displacement.

8.5. Entity Class – Known Procedure

8.5.1.1. Encode Call

The emitter emits a CALL instruction which calls near, relative, displacement relative to next instruction using the display register, register and the address displacement.

8.5.1.2. Encode Fetch

The emitter first emits a LOADA instruction which pushes the provided address onto the stack using the frame display register. The emitter then emits the LOADA instruction using the register and address displacement.

8.6. Entity Class – Known Value

8.6.1.1. Encode Fetch

The emitter emits a LOADL instruction which pushes the provided literal value onto the stack using the provided value.

8.7. Entity Class – Unknown Address

8.7.1.1. Encode Assign

The emitter first emits a LOAD instruction which loads an object from the provided address and pushes in on the top of the stack, using the machine address size, display register and address displacement. The emitter then emits a STOREI instruction which pops an address and an object from the stack and stores it at that address.

8.7.1.2. Encode Fetch

The emitter emits a LOADI instruction which pops a data address from the stack, fetches an object from that address and pushes it onto the stack, using the machine address size, display register and address displacement. The emitter then emits a LOAD instruction which loads an object from the provided address and pushes in on the top of the stack, using the machine address size, display register and address displacement.

8.7.1.3. Encode Fetch Address

The emitter first emits a LOAD instruction which loads an object from the provided address and pushes in on the top of the stack, using the machine address size, display register and address displacement.

8.8. Entity Class – Unknown Value

8.8.1.1. Encode Fetch

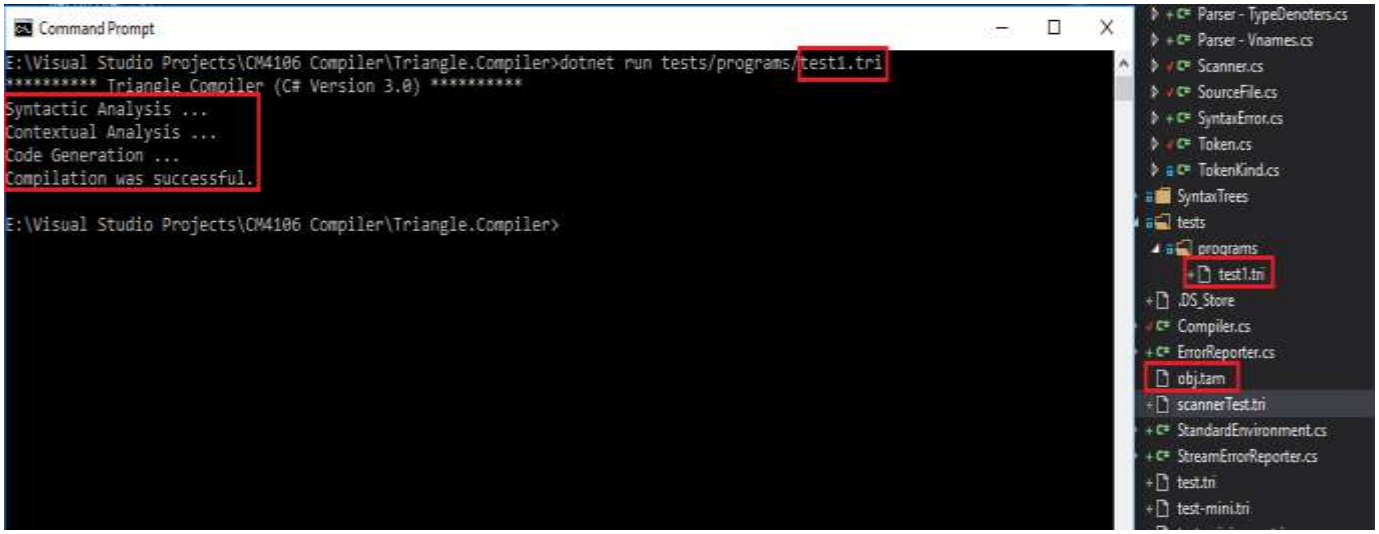
The emitter first emits a LOAD instruction which loads an object from the provided address and pushes in on the top of the stack, using the frame size, display register and address displacement.

9. Test Cases

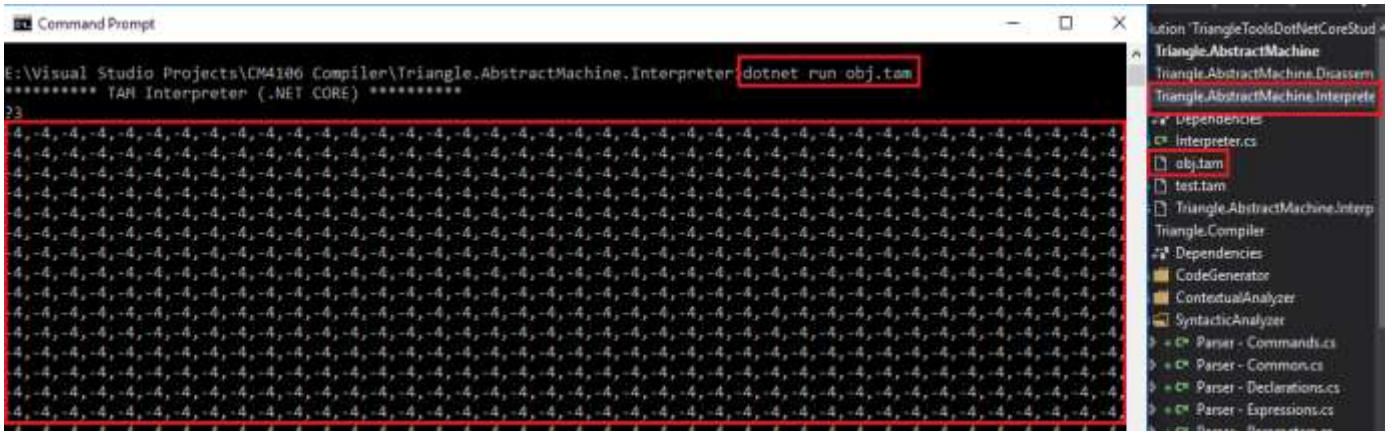
To test the compiler, first a test file needs to be executed to make sure that the compiler can create the obj.tam file which will be executed by the interpreter. If the compiler fails to the an obj.tam file, then part of the functionality is not working as intended.

9.1. Code Generator Test

The first test is to check whether the compiler can produce an obj.tam file if all three checks have been passed successfully. The picture below shows that all three checks that include the Syntatic Analysis, Contextual Analysis and the Code Generation have been successful when running the test1.tri file. Since the Compilation process was successful, an obj.tam file has been created that is going to be executed by the interpreter.

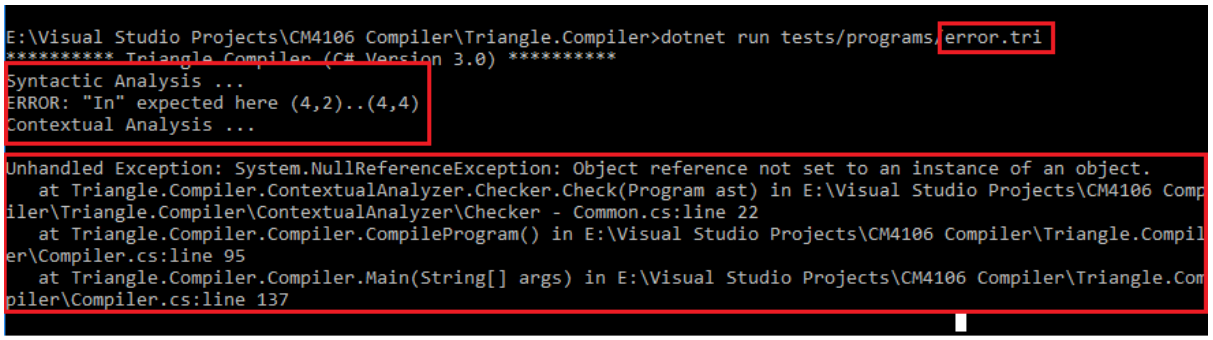


The next step is to copy the obj.tam file into the interpreter to check whether the file can be executed successfully. The picture below shows that the file is now in the interpreter and it has been executed successfully, but because how the program works, no matter what value is given as an input, the program will loop infinitely producing -4.



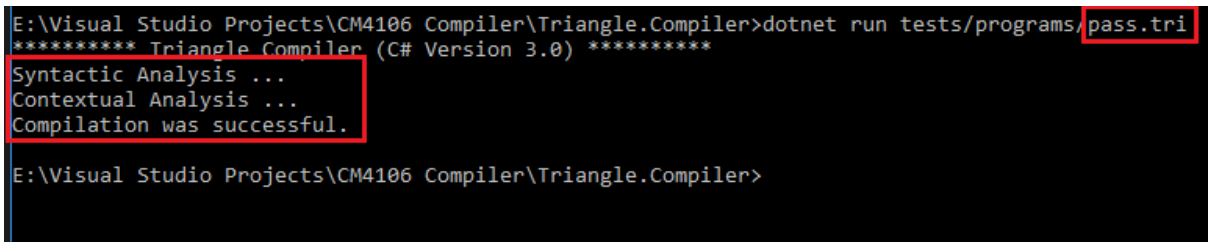
9.2. Semantic Analyser Error Test

The picture below shows a test using a file that has errors to check whether the semantic analyser can detect them. As it is shown on the picture, the syntactic check is passed, but the semantic analyser finds an error in the code. Unfortunately, this also causes dotnet to crash most likely, because of how the semantic analyser returns values whenever an error is found. The semantic analyser will return null in the case of an error which causes the annotated abstract syntax tree building to fail.



9.3. Semantic Analyser Pass Test

The final test, checks whether the semantic analyser can successfully check a file that has no errors. The picture below shows a file being run only through the syntactic and semantic analysers and displays no errors which shows that the test has been successful.



10. Code Generator Test File

```
let
   const MAX ~ 100;
   var y: Integer;
   var x: Integer
in
begin
   y := 5;
   put('?');
   getint(var x);
   if       (x>0) /\ (x<=MAX) then
                        while x > 0 do
                        begin
                                 x := x-y;
                                 putint(x);
                                 put(',')
                        end
            else
end
```

11. Semantic Analyser Error Test File

```
let
  const MAX ~ 10;
  var n: Integer
  if (n>0) /\ (n<=MAX) then
    while n > 0 do begin
      n := n-1
    end
    else
end
```

12. Semantic Analyser Pass Test File

```
let
  const MAX ~ 10;
  var n: Integer
in
begin
  if (n>0) /\ (n<=MAX) then
    while n > 0 do
            begin
                     n := n-1
            end
    else
end
```