Jérôme Charrière
Student ID: 40003247

COMP 6421: Assignment #4

Winter 2017

Concordia University

Table of content

1. Semantic of the language

1.1. Description of the peculiarities of the language

The following section documents information on the language which may not be intuitively known by just analyzing the grammar.

1.1.1. Functions

1.1.1.1. Function return type

The return types of functions can be either integers, float numbers and user-defined types (objects). However, the program does not allow arrays (of any types) to be returned.

1.1.1.2. Function parameters

Parameters are allowed to be either: integers, float numbers and user-defined types (objects). However, the program does not allow the declaration of arrays (of any types) as parameters.

1.1.1.3. Function instances

Due to the implementation of a stack, there can be any numbers of function instances at the same time.

1.1.2. Assignment statements

1.1.2.1. Assignment types

Variables can be assigned values that match their types. Depending on the type of the variable: objects, float and integers may be assigned. However, the program does not allow arrays (of any types) to be assigned, even if the variable being assigned was specified as an array (e.g “int a[4]; int b[3][5]; a = b[0];” is not valid).

1.1.3. Expressions

1.1.3.1. Expressions for put statement.

“put” expects an expression as its value. To be deemed valid, the evaluated expression must be an integer.

1.1.3.2. Expressions for get statements.

“get” expects a variable as argument. Because the captured input by “get” can only be an integer, the type of the variable expected has to be an integer.

1.1.3.3. Relational expressions for conditional statements.

Relation expressions can accept either floats and integers on the left and right sides of the relational operator.

1.1.3.4. Expressions representing array indexes.

Evaluated values of expressions representing array indexes must be of type integer.

1.2. Grammar augmented with semantic actions

#CREATE_GLOBAL_TABLE	Create a global table and set it as current table scope (subsequent entries will be stored in this table until #SCOPEUP is reached)
#CREATE_CLASS_ENTRY_AND_TABLE	Create an entry of type “class” in the current table scope; create a table and set it as current table scope (subsequent entries will be stored in this table until #SCOPEUP is reached)
#CREATE_FUNC_ENTRY_AND_TABLE	Create an entry of type “function” in the current table scope; create a table and set it as current table scope (subsequent entries will be stored in this table until #SCOPEUP is reached)
#CREATE_VAR_ENTRY	Create an entry of type “variable” in the current table scope.
#CREATE_FOR_TABLE	Create a table and set it as current table scope.
#ADD_PARAMETER_TO_FUNC_TABLE	Add parameter information to function entry.
#SCOPEUP	Change the current scope to the current scope's parent table.
#TYPE_CHECK(int[] types, AbstractSyntaxTree tree)	Type-check the given expression tree (second argument) based on a set of expected types (first argument).

```
prog            -> #CREATE_GLOBAL_TABLE classDecl_N progBody
classDecl_N -> classDecl classDecl_N | EPSILON
classDecl       -> 'class' 'id' #CREATE_CLASS_ENTRY_AND_TABLE
'{'varDecl_N_or_funcDef_N'}'';' #SCOPEUP
varDecl_N_or_funcDef_N -> varDecl_N_or_funcDef varDecl_N_or_funcDef_N | EPSILON
varDecl_N_or_funcDef -> type 'id' varDecl_N_or_funcDef_tail
varDecl_N_or_funcDef_tail -> arraySize_N ';'  #CREATE_VAR_ENTRY |
#CREATE_FUNC_ENTRY_AND_TABLE '(' fParams ')'  funcBody ';' #SCOPEUP
```

```
varDecl_N -> varDecl varDecl_N | EPSILON
funcDef_N -> funcDef funcDef_N | EPSILON
progBody        -> 'program' #CREATE_FUNC_ENTRY_AND_TABLE funcBody ';' #SCOPEUP
funcDef_N
funcHead        -> type 'id' #CREATE_FUNC_ENTRY_AND_TABLE '(' fParams ')'
funcDef_N -> funcDef funcDef_N | EPSILON
funcDef         -> funcHead funcBody ';' #SCOPEUP
funcBody        -> '{'varDecl_or_statement_N'}'
varDecl_or_statement_N -> varDecl_or_statement varDecl_or_statement_N |
EPSILON
varDecl_or_statement -> 'id' varDecl_or_statement_tail | type_
varDecl_tail | statement_other
varDecl_or_statement_tail -> varDecl_tail #CREATE_VAR_ENTRY | indice_N idnest_N
assignStat_tail ';'
assignStat_tail -> assignOp expr
statement_other -> 'if' '(' expr ')' 'then' statBlock 'else'
statBlock ';'
                | 'if' '(' expr ')' 'then' statBlock 'else' statBlock ';'
#TYPE_CHECK({int}, expr_tree)
                | 'for' '(' type 'id' assignOp expr ';' relExpr ';' assignStat ')'
statBlock ';' #TYPE_CHECK({type}, expr_tree)
                | 'get' '(' variable ')' ';' #TYPE_CHECK({int}, variable)
                | 'put' '(' expr ')' ';' #TYPE_CHECK({int}, expr_tree)
                | 'return' '(' expr ')' ';' #TYPE_CHECK({function_type}, expr_tree)
idnest_N        -> idnest idnest_N | EPSILON
idnest          -> . 'id' indice_N
varDecl_tail -> 'id' arraySize_N ';'
type_ -> 'int' | 'float'
varDecl         -> type 'id' arraySize_N ';' #CREATE_VAR_ENTRY
statement_N -> statement statement_N | EPSILON
statement       -> assignStat ';'
                | 'if' '(' expr ')' 'then' statBlock 'else' statBlock ';'
#TYPE_CHECK({int}, expr_tree)
                | 'for' '(' type 'id' assignOp expr ';' relExpr ';' assignStat ')'
#TYPE_CHECK({type}, expr_tree)
statBlock ';'
                | 'get' '(' variable ')' ';' #TYPE_CHECK({int}, variable);
                | 'put' '(' expr ')' ';' #TYPE_CHECK({int}, expr_tree);
                | 'return' '(' expr ')' ';' #TYPE_CHECK({function_type}, expr_tree)
assignStat      -> variable assignOp expr #TYPE_CHECK({variable_type}, exp_tree)
statBlock       -> '{' statement_N '}' | statement | EPSILON
expr -> arithExpr expr_tail
relExpr         -> arithExpr relOp arithExpr
expr_tail -> relOp arithExpr | EPSILON
arithExpr       -> term arithExpr_rr
arithExpr_rr -> addOp term arithExpr_rr | EPSILON
sign            -> '+' | '-'
term            -> factor term_rr
term_rr         -> multOp factor term_rr | EPSILON
factor          -> variable_or_functionCall
                | num
                | '(' arithExpr ')'
                | 'not' factor
                | sign factor
variable_or_functionCall -> id indice_N idnest_N variable_or_functionCall_tail
variable_or_functionCall_tail -> '(' aParams ')' | EPSILON
variable        -> 'id' indice_N idnest_N
indice_N -> indice indice_N | EPSILON
indice          -> '[' arithExpr ']'
```

```
arraySize_N -> arraySize arraySize_N | EPSILON
arraySize       -> '[' 'int' ']'
type            -> 'int' | 'float' | 'id'
fParams         -> type 'id' arraySize_N #ADD_PARAMETER_TO_FUNC_TABLE fParamsTail_N
| EPSILON
aParams         -> expr aParamsTail_N | EPSILON
fParamsTail_N -> fParamsTail fParamsTail_N | EPSILON
fParamsTail     -> ',' type 'id' arraySize_N #ADD_PARAMETER_TO_FUNC_TABLE
aParamsTail_N -> aParamsTail aParamsTail_N | EPSILON
aParamsTail     -> ',' expr
assignOp        -> '='
relOp           -> '>' | '<' | '<=' | '>=' | '<>' | '=='
addOp           -> '+' | '-' | 'or'
multOp          -> '*' | '/' | 'and'
num             -> 'int' | 'float'
```

2. Architecture description

2.1. Changes added to the semantic analyzer

2.1.1. Type checking

2.1.1.1. Structure of the type checking.

Mainly, the semantic analyzer was augmented with the introduction of type checking. Type checking verifies that the type of elements within an expression match certain type(s). The definition of those matching types are based on the kind of type checking at hand. The type of operations will be detailed in the next section.
Also, the type-checking process now include other (previously and new) defined semantic checks (such as checking if a variable/function is declared, number of parameters passed if function, number of dimensions etc..).

To represent expressions, the syntactic analyzer uses attribute migration to build an abstract syntax tree. Abstract syntax trees are built exclusively for the representation of expressions (everything included by the <expr> rule in the original grammar.). The elements found at the leaves of the tree can be either: constant values, function calls and variables.

To type-check an expression tree, the semantic analyzer will traverse the tree in post-order. If the type of an element is found not to match the expected type, or additionally, if the element is found not to be valid (ex: undeclared variable), an error will be reported but the analysis will continue. Hence, the type-checking won't stop at the first error encountered and will be able to analyze every single elements in the expression.

2.1.1.2. Kinds of type checking.

- If we seek to type-check an assignment statement, then the expected type would be the type of the variable on the left-hand side of the assignment.
- If we seek to type check a return statement, then the expected type would be the type of the

value returned by the function.

- For putc, the expected type of the elements passed would be integers.
- For getc, which can take a variable, the expected type would be an integer.
- For relational expressions, which are used by “for” statements, The expected types are multiple. We expect either a float or an integer. This is due to the presence of a relational operator which will cause the expression to produce an integer (boolean value). Hence, floats can be allowed in this context.
- For array indices, we can only accept integers as values. Hence integers are the expected type.

2.1.1.3. Parameter checking

When a function call is being analyzed, we will now verify if the function call parameters are valid.

To do so, we first check the number of parameters passed. If this number is not equal to the expected number of arguments then we will simply output an error and we will not proceed any further.

Otherwise, if this condition is satisfied, then we will begin to type-checkS each of the arguments. The type-checking uses the same strategy as the type-checking highlighted in the section above. For each of the expression arguments, we verify if the types of the elements at the leaves of the expression are of the same type as the type of the parameter initially declared.

2.1.1.4. Dimension checking

When analyzing a variable, or a method call: for each of the identifier in the identifier nest, we verify if array dimensions were specified in the identifier declaration. If so, we verify if the reference provided array indexes and if the number of indexes equals the number of array dimensions initially declared.

2.2. Structure of the code generator

After the successful lexical/syntactical and semantic analysis, the compiler will then proceed to generate code.

To do so, the code generator make available to the syntactical analyzer, a set of methods that can be called within parsing functions.

Those methods will be called by the syntactic analyzer during a third pass, where it will parse the program once more. As an example, when the syntactic analyzer will encounter an assignment statement, it will gather an attribute representing the variable on the left-hand side, and an abstract syntax tree to represent the expression on the right hand side of the assignment statement. After doing so, the syntactic analyzer will pass these data to a method of CodeGenerator.

Overall, most of the logic inside the code generator was focused on the evaluation of expressions. The following sections will describe several of the methods used to evaluate expressions and output code.

2.2.1. The getValue() method

The getValue() method takes an abstract syntax tree node (more precisely: a leaf) as argument and will evaluate the element attached to it. The element we can find at a leaf may be of different types. Namely, it can be a constant value, a function call or a variable. Different behaviors are implemented

depending on the type of the element encountered.

2.2.1.1. Encountering a constant value

If we have encountered a constant value, getValue() only needs to load this value into the register id passed as argument.

2.2.1.2. Encountering a variable

When encoutering a variable, getValue() will first have to calculate the address of the variable before loading its value into the designated register. To do so, the method uses another helper method getAddress(). As variables can be expressed in the form of a path of different objects which also may include array indices, the getAddress() method will have to calculate the address (offset) of each of the elements that make up the variable path while accounting for arrays as well. However, the method will not return an integer value representing an address or an offset. Indeed, the presence of arrays within variable paths prevent the code generator to compute the exact location where the element is, as array indexes are represented in the form of expressions that also need to be evaluated. This mean that we cannot calculate variable addresses at compile-time and that those calculations must be done so at run-time. To take care of this scenario, the getAddress() method will output assembly code that will perform this calculation. Hence the returned value to getValue() is a register that will be holding, in the end, the address (offset, more precisely) of the inputted variable.

2.2.1.3. Encountering a function call

If a function call was encountered and if we detect that the function call is a method call, we will have to perform the same strategy as variable address resolution. Indeed, we need to know the address of the object's instance so that the address will be loaded into a special register. Hence, enabling the callee method to interact with its instance's attributes by using this value inside this special register as an offset. However, if the function is a free function, there will be no need to do all those operations.

The code generator uses a stack based approach for memory. Whenever a function is called or returning, there is a set of operations that the code generator will have to output.

2.2.1.4. Calling a function

As for when a function is being called. The program will have to output code to allocate space on the stack for the returned value, save the general registers on the stack, save the current stack frame pointer, save the current function return address and object instance reference addresses held in registers.

After doing so, then the code generator will proceed to setup a new stack frame pointer value, using the current stack pointer as the new pointer. It will then finally evaluate each of the function parameters passed as arguments and place them accordingly on the stack so that the callee will be able to fetch them using the computed offsets during the semantic analysis.

2.2.1.5. Returning from a function

As for returning from a function: Before jumping back to the code of the caller function, we need to perform several operations. First, we need to setup the return value. This is done by copying the targeted value on the callee's stack to the return space setup by the caller. Note that in the case of returning objects, the code generator will output code to perform a copy of the object. Afterward, we need to restore the previous function context. We will restore the registers, both specials and generals. Finally, the function that initiated the call will find the returned value where the current stack pointer sits.

2.2.2. The evaluate() method

The evaluate() method should be called when we seek to evaluate an expression that can take an object as part of the expression's elements and when we want the computed expression value to be outputted at a certain address rather than receive it directly into a register. This method is useful for assignment, return, put and get statements as all expressions within those statements need the evaluated values to be saved at specific locations in memory.

2.2.3. The mathEvaluate() method

The mathEvaluate() method should be called when we need the value within an expression to be contained within a register rather than saved on the stack.

3. Tools used

3.1. Moon machine's debugging tool

If the tracing option is inputted to the Moon machine when executing an assembly program, the Moon machine will enter in a tracing mode. This enable the debugging of programs, where users can put breakpoints at specific location in the generated code, or see the flow of exeuction by executing 10 instructions each steps. For each of the instruction displayed, the Moon machine show the values inside the registers used and the location in memory where a value has been saved/loaded in the case of an instruction maniuplating memory.