Maciej Grabowski   Lab 7 – Report   12304093

Multi-Dimensional Arrays

Syntax

```
int myArray[5][5];
```

The above code initializes a 2-dimensional array and sets all its elements, 25 elements in this case, to store value 0.

```
myArray[c][r] := 1;
```

The above code sample sets the c's element's value of the r's dimension to 1.

```
myArray[c][r]
```

The above syntax is used for referencing elements of already initialized 2-dimensional array.

The above examples refer explicitly to 2-dimensional arrays however this syntax holds for any number of dimensions for example to initialize a 4-dimensional array you would use the following syntax:

```
int myArray[m][n][o][p];
```

where m,n,o,p are integer constants or variables specifying the size of particular dimensions.

Implementation

Initialization

To implement the initialization of the multi-dimensional arrays the variable declaration production rules have been modified. Apart from allowing for Type followed by Ident non-terminals and a semicolon I allowed to optionally include square brackets after the identifier which would indicate that the user is initializing an array if the correct syntax is used. Therefore the compiler makes a decision of whether an input is a variable or an array depending on what comes next after the identifier and then appropriate code gets generated.

When the compiler comes across the array initialization it reserves space for all the elements on the stack of Symbols. Each element is associated with an appropriate name where the name of the array is followed by a set/sets of square brackets specifying an unique index within those brackets, for example:

```
int myArray[3];
```

Stack of symbols:

```
Symbol("myArray[0]", ...);
Symbol("myArray[1]", ...);
Symbol("myArray[2]", ...);
```

This works similarly for multi-dimensional arrays where each element creates a symbol and pushes it onto the stack of symbols at the consecutive indexes in the current frame.

<u>Element Assignment</u>

The element assignment involved modifying the Stat production rules. It works similar to variable assignment however the name with which each of the array elements are associated is a little bit different since a correct identifier specifies the array name on its own but it was necessary to optionally expect a set of square brackets after the identifier and added them on to the name together with the indexes because of the way the elements are stored in the memory. The rest of the assignment is handled exactly the same as a usual variable assignment.

<u>Accessing Elements</u>

Accessing of array elements is implemented very similarly to variable accessing. The only difference is the name that is associated with a Symbol of each element. The name is made up of an identifier and then is followed by a set of square brackets specifying indexes. In order to implement this the Factor production rules were modified.

<u>Handling Arrays Out Of Bounds</u>

Handling accesses to elements that are out of bounds is really simple in my implementation of the compiler since each element is stored on the stack of symbols and each element has its own name that is associated with. If somebody tries to access an element that is out of bounds and hence had not been initialized it will result in an error at compile time as the symbol associated with the element does not exist on the stack.

Switch Statement

Syntax

<u>Note</u>: Switch works only with integer values.

```
switch(x){
  case 0{
    //some code here
    break;
  }
  case1
  case2
  case3{
    //some code here
    break;
  }
  default{
    //some code here
  }
}
```

The above is a sample code where an integer variable x previously initialized and assigned is switched. The code will execute a given case that meets and *Equ* condition between the integer variable x and an integer expression following the *case* keyword. The break statement is used to break out of the switch statement when it's not required to go through the following cases. The statement default may be specified and will be executed if no case condition is met however it is not mandatory to include it in the switch statement.

Implementation

The switch statement is part of the statements(Stat) production. It is identified by the *switch* keyword followed by an opening bracket followed by an integer variable and a closing bracket. The body of the switch statement must be enclosed by the curly brackets and may include cases and/or default. Case is identified by the keyword case followed by an expression of type integer and may be followed by a Stat production. Default is identified by the keyword default and must be followed by a Stat production.

When the switch statement is encountered by the compiler the switch variable's Symbol is retrieved through its identifier and is temporarily stored for later referencing within the body. When a case is encountered within the switch body the two *Const* instructions are added to the program in order to push the value specified by the case as well as the value pointed to by the switch variable on the VM's stack. Next the *Equ* instruction is added which will compare the last 2 values from the stack and will push the result on to the stack. Next a label is generated and its later placed at the end of the case after its optional body or immediately after the case if no body has been defined. Next *Fjmp* instruction, pointing to the just generated label, is added to the program and it makes use of the last value pushed on to the stack by the *Equ* instruction. If the instruction produced truth then the program will proceed to execute the body of the case and if the *Equ* instruction has produced false then the program will jump to the next case omitting the body of the current case.

The *break* was implemented as a separate statement to switch as it would make it easier to allow for its use within loops as well(however this feature was never implemented at the end). For clarity and to avoid collisions(break statements popping non-breakable statements) the new stack of labels was implemented within the compiler that stores all the current open breakable statements.

Structure

Syntax

<u>Declaration</u>

```
Struct Rectangle{
  int width;
  int height;
  int area;
}
```

The above is a sample code of C-like structure that has been implemented as Tastier's feature. The structures may only be defined before any functions and within the program. They are created using the *Struct* keyword followed by an identifier. The *Struct* requires a body and it must be enclosed with curly brackets. Within the structure's declaration it's only possible to initialize variables.

Initialization

```
Rectangle rectA;
Rectangle rectB;
```

The above sample refers to previous example of structure declaration where Rectangle structure was declared. Tastier allows for multiple instances of the same structures where each instance is independent of one another. After the initialization of an instance we can access each of its members.

Accessing and Assigning Structure's Elements

```
rectA.width := 5;
rectA.height := 4;
rectA.area := rectA.width * rectA.height;
```

After the initialization of a structure we can access its members by specifying the structure's instance's identifier followed by a dot and the member's identifier as per the above example.

Implementation

In order to implement the structures new production has been added on to the grammar, namely StructDecl. The StructDecl appears in the program production together with constant declarations before any procedure declarations. This ensures that structures have to be defined before procedures.

The StructDecl production begins with Struct keyword and an identifier non-terminal. The production also states that the body of the structure has to be enclosed by curly brackets and may only contain variable initializations. The declarations of structures are stored on the stack of symbols at the compile time where symbols of each member of the structure are associated with a name that is made up of <structure_identifier>.<member's_identifier>, for example:

```
Struct Rectangle{
  int width;
  int height;
  int area;
}
```

Stack:

```
...
Symbol(Rectangle.width, ...);
Symbol(Rectangle.height, …);
Symbol(Rectangle.area, ...);
...
```

Therefore when initializing an instance of a structure new lookup function was introduced in the compiler that looks through the open scopes of the stack for Symbols containing a dot, compares the structure identifiers through string manipulation and returns an array of all the symbols that are members of the structure in an identified scope. This allows to create new Symbols that will be referring the instance's structure members and will be treated as normal variables like for example:

```
Rectangle rectA;
```

Current stack:

```
Symbol(rectA.width, …);
Symbol(rectA.height, …);
Symbol(rectA.area, …);
```