CS3071 – Exercise 7 Report

Edward Bergman
#15335633

Task:

For this excersize, we had multiple goals. We implemeneted For loops, Structs. We have previously implemented Runtime bounds checking of arrays which was told to be satisfactory for this assignment. We also additionally added break statements to the Tastier language.

Prelude:

Major changes were made to the ATG and symbol table. We will not go into these changes in detail as this would require reports in themselves.
The most notable forms of changes revolve around:

Restructing the ATG – We moved most things to be some form of statement. For example, variable declarations, proc declerations, const declarations, reads, writes etc.. are now all Stat (statements). In conjunction with this, instead of having specific locations for procs, variable declarations and statements, now that everything is a statement, all things can exist wherever they wish. This allows for a more flexible language structure. The ATG was also reorganized to have as much seperation in the rules as possible (to eliminate the hundreds of lines long multiple choice Production Rules). To help understand the ATG we recommend viewing the file in the following order :

1. *At the bottom of the file, you can see the program structure which is mainly a program with some statements.*
2. *Each of these statements will try as best as possible to have their own seperate production.*
3. *The components that make up a Statement are listed at the top of the ATG (i.e. Primary, Expr, Ident...)*

Restructuring the Symbol Table's Type system – This will be explained in more detail in the Struct section but the Type system in place was extended so we could encode more information. The variable mapped to an int was not expressive enough to capture the idea of a struct (Which we treat as a type). The type system was revamped to now have Type's as objects with an id and size information. This allowed us to be more flexible and add types dynamically during compile time. For further informormation, refer to Type.cs and the 'types' Dictionary in the SymTab.cs file.

***For Loop*:**

Specification:

The for loop is a simple structure designed for iteration. To provide for this iteration, our for loops shall allow some kind of initilization, a conditional check upon each loop and a simple statement to be performed upon each iteration of the loop.

Method:

Firstly, as with the other exercises, we must begin by designing a regular expression to match a For loop statement.

```
ForStat = “for” ‘(‘
            [ Ident { StructField } VarAssignment ]
            [ Expr ‘;’]
            [ SimpleStat ] ‘)’
          ‘{‘
            { Stat }
          ‘}’
```

Breaking this down, our optional variable assignment is some variable followed by a struct field followed by an assignment to this variable. (We will StructField later but it is for specifiying fields of a struct object) VarAssignment is a production rule that factored out the ‘:=’ assignment operator but essentailly assigns a value to some Var. (See Tastier.ATG::VarAssignment for details)

Our Expr is an optional boolean conditional expression which controls when the loop will exit. If this is not present, the loop will continue endlessly unless we execute a BreakStat (We will cover this later).

Lastly, our SimpleStat is the third optional we can specify. We can see which kinds of statements are allowed near the bottom of our ATG but they are Stat’s that do no provide any deeper program structure other than expressions. It does not make much sense to allow another ‘ForStat’ or ‘SwitchStat’ in this optional.

The next step was control the program flow of the ForStat, the first step to accomplising this was to establish the different places we had to jump to.

```
‘for’ ‘(‘ Assignment
        <conditionLabel> Condition
        <postStatLabel> PostStat ‘)’
      ‘{‘
           <statLabel>
            Statements
      ‘}’
      <endLabel>
```

Conditionlabel as we must check this multiple times.
PostStatLabel as we execute this statement multiple times.
StatLabels as we have to jump past the PostStat after checking the condition.
EndLabel as we have to be able to exit the loop.

Our flow goes as such

```
<conditionLabel> -> Condition check ( <statLabel> | <endLabel> )
<postStatLabel> -> PostStat <conditionLabel>
<statLabel> -> Statements <postStatLabel>
```

We also scope the for loop such that all symbols created in the for loop will only exist in this loop. This is also vital for our BreakStat to work as we shall see once we elaborate.

The last detail of the implementation that is noteworthy is that we have to anticipate the scenario of no conditional expression being specified. To handle this, we model Java’s for loop style of simply iterating continously unless we

break out of the loop. To implement this, we simply use LoadTrue to set the flags accordingly, if a condtional expression is present, this will overwrite the flags.

Remarks:

The ForStat which specifies the for loop is relatively straight forward. The main challenge was identifying the correct program flow and implementing it as the code will not be generated in the correct order to be sequentially executed. We also had to be careful to limit the possible set of Statements possible in PostStat. The choice regarding which Statements were allowed boiled down to which Statments fit nicely into one line. In contrast, we did not want a While Loop to be possible in the PostStat where as something as simple as a procedure call, assignment or some read/write function was considered okay. This was more of an arbitrary choice based on preference and to encourage a programming style of simplicty and for the legibilty of a ForStat.

Learnt:

While not much was learnt from simple the ForStat in itself, to implement the ForStat in the manner we desired, a large restructing provided greater flexibilty and a greater understanding of the different kinds of program structures and how the interact within a language. This understanding was achieved while deciding how we wanted to fit all these components together.

We also learnt a good method for how to plan a ‘control’ programming structure. (control programming structure being some programming structure that does not follow a sequential execution pattern but requires code jumping). This method is to identify the different components of the structure and how many times these components will be executed and plan for label placements accordingly.

Break Statement:

Specification:

A break statement will exit the current ‘control’ programming structure where appropriate.

Method:

A break statment fufills a very basic purpose which is to jump to some point in the code. The issue is identifying the ‘appropriate’ structures where this is valid and also where to jump to. We combat these together by using the functionality already provided in the Symbol Table which is the Open/Close SubScope methods provided.

We established the rule that Sub Scopes would be used for the purpose of specifying a scope specific to a ‘control’ programming structure. These are often structures that rely on conditionals to control program flow. This allows us to make the statement that a BreakStat will be used to jump to the end of the subscope.

The second issue about where to jump to could be tackled in two ways. One is to pass some compile time information through Stat to BreakStat on where this end label is. We did not choose this method as most statements do not require this knowledge of an end label and hence added extra noise to our ATG.

We instead took the approach that once you open a sub scope, you pass information to the symbol table about the endlabel. We can do this by calling *tab.OpenSubScope(endlabel).* This allows any occurence of BreakStat to check in the symbol table if it is placed within a valid SubScope.
We gain two benefits from this method :

1. We no longer have to pass information through Stat to the BreakStat.
2. We can check if the BreakStat is in a valid scope to be broken out of.

Remarks:

The BrakeStat was implemented as there are situations where we wish to cease processing in a loop at an earlier time.
For example, suppose our loop is iterating through a list to find a number in a list of size 1,000,000. We wish to stop iteration once we have found our number.

Without our BrakeStat, we will be forced to use the conditional variable to end interation. In a loop where we might check for multiple conditions for exit, this can add a lot of noise to the For Loop declaration and make the logic of the loop harder to read.

With the BrakeStat, “break;” provides a clear semantic way to specifiy we wish to stop execution here. This allows us to leave the conditional to remain clear that we will loop to 1,000,000. Following the logic of the loop itself, we can clearly see where execution should cease if required.

Lastly, we made it a Statement in itself as we usually require it in places where Statments are made. (i.e. inside an if/else statement)

Learnt:

The principle behind a break statement is relativly straight forward, the actual implementation of this idea requires some identification of which kind of programming structures should be allowed to be broken out of. We had to learn how to identify these structures and how close the link is between scopes and these structures. Once we learnt how they interact, this allowed us to identify where a break statement fits into the picture.

Structs:

Specification:

You can create a struct definition which will define how structs of this type will be structured. This definition will consist of a Type identifier and fields. A Field is an identifier with an associated Type. This Type can be another struct. You can create multiple struct objects which are of of the Type of their definition.

We wished to allow for some more features to our structs for greater flexibilty but for time reasons, we have chosen to implement only the above.

Method:

Type System

There was quite a lot to be done to implement Structs as part of Tastier's Type system. Structs require a degree more information to define them which the current type system consisting of a variables mapped to ids did not allow for.

The first step we took was to identify what information we needed to create a definition of a Struct such that it could be used as a type. The following pieces of information were needed :

1. A name to identify the Struct Type.
2. A size so we can allocate the appropriate stack space.
3. A numerical Id which allows them to fit into the current system.

Part 3 was not strictly neccessary as we could have redone the ATG's type checking system. To keep things quick, single integer comparison is faster than string comparison (using names for identifiers). It was also deemed easier to integrate into the existing system rather than reimplement type checking. Refer to *Type.cs* for a full view of the new definition.

With this new definition of a Type, we converted the old primitive types to the newer definition of a Type. We also used a dictionary to store these types and allow for fast lookup of types based of id.

```
//types
private static int nextTypeId = 0;
public Dictionary<int, Type> types = new Dictionary<int, Type>()
{
    { 0, new Type("undef", -1, nextTypeId++) },
    { 1, new Type("int", 1, nextTypeId++) },
    { 2, new Type("bool", 1, nextTypeId++) }
};
```

With this new Type system in place, we could now move onto our next objective, defining a new Type which is associated with a Struct Definition. These will be linked by their TypeId and their name.

Struct Definition

A Struct Definition is a purely compile time construct. As such, all information regarding how a struct is structured is present in the Symbol Table.

To help seperate concerns, we made a new file for Struct Definitions, namely StructDef.cs. A struct definiton needed specific information to be able to map our fields to specfic addresses. As such, we can define a Struct as a List of Fields with specific offsets into the Struct's space on the stack.

Each field contains the following information.

```
public int offset;
public String name;
public Type type;
```

Knowing all fields that occur in a Struct Definition allows us to calculate the space required for every instance of this struct. (Refering to type for info) For complete information on a StructDef, please refer to StructDef.cs.

Now that we can store information required to store a struct instance based on a struct Type, we can now finally begin to modify our ATG and implement a production rule to create one of these Struct Definitions.

```
StructDecl              (. string structName, name; int type; .)
=
"def" "struct" Ident<out structName> '{'  (. List<Tuple<Type, String, int>> fields
                                               = new List<Tuple<Type, String, int>>(); .)
  {
    Type<out type> Ident<out name>        (.  Type t = tab.getType(type);
                                              fields.Add( new Tuple<Type, String, int>(t, name, 1));
                                           .)
      { ',' Ident<out name>               (.  fields.Add( new Tuple<Type, String, int>(t, name, 1)); .)
  }';'
  }                                       (.  tab.newStructDef(structName, fields); .)
'}'
.
```

With all the heavy lifting taking place in the Symbol Table, the StructDecl is very straight forward. We would have seperated these explanations but we feel the production rule is fairly straight forward.

We take an Ident to get the structName, our name for the new Type. Then, for each field specified (Type Ident) we create a new tuple storing three pieces of information, type, field name, size. We will discuss this further later but all scalars have size 1 where as arrays would have a size > 1. Unfortunatly, we did not implement arrays as a field in our struct but we shall discuss this in our remarks.

These Struct Definitions are stored in a Dictionary in our symbol table, mapping a struct's name to its definition.

As a side benefit of this implementation, we could also implement scoping definitions. By keeping track of the scope of where the definitions we made, we could remove these definitions from the dictionary once we closed the scope. This makes sense as a struct defined in one procedure shouldn't be available in another procedure.

Struct Instances

Our last goal is to implement how structs are instansiated and used. To instansiate a struct was quite straight forward. As our structs are considered to be Types, an instance of a struct is the same as a variable declaration. In the same way we can say 'int myInt', we can say 'struct MyStruct foo'. This was done by modifying the Type production rule.

```
Type<out int type>
=                           (. type = undef; .)
 ( "int"                    (. type = integer; .)
 | "bool"                   (. type = boolean; .)
 | "struct"                 (. String structType; .)
    Ident<out structType>
                            (. type = tab.getStructDef(structType).typeId; .)
 )
.
```

The only difference is that we do not allow for an array of structs as we allow for int and bool. The reasons for this is discussed in remarks. We did this by checking if the type was a struct or a primative in our VarDecl production rule. We implemented a small method in the symbol table for this,

typeIsStruct(int typeId)

Now that we can specify which kind of Struct we wish to instansiate, we must allocate the proper space on the stack. We fortunatly know the space required as we have this stored in our StructDef. We also know the offset required for each field.

We chose to implement each field in a struct instance as its own object. This allowed us to give each field its own unique name in the symbol table as well as an associated address. We could garuntee each name was unique as the first part of the name was unqiue (i.e. “foo.value” is seperate from “bar.value” even if foo and bar are of the same Type).

The issue we faced with this method was that we needed to modify the NewObj method such that we could specify the address of these objects. If we did not, we would allocate space for the struct object and then as we try to instansiate the fields as objects, they would be just given the next free address on the stack and not the address within the space allocated for the struct.

```
Next free address ->    | -------
            { Foo.a     | address1
              Foo.b     | address2
      Foo ->{ Foo.c     | address3
```

As shown in the example above, we needed to give the unqiue object Foo.a the address ‘*address1*’ which was already allocated for the Foo object. This is why we changed NewObj to take in an address parameter.

Now that we could instansiate a struct type object, our last goal is to be able to access these fields. As each field was in itself its own object, it was quite easy. All we needed was some production rule that would allow us to specify these objects. This was done with the rule StructField.

```
StructField<string pName, out string name> (. String fieldName; .)
=
'.'
Ident<out fieldName>          (. name = tab.Find(pName + '.' + fieldName).name; .)

.
```

As we can see, this is relativly simple, it concatenates identifiers seperated by a ‘.’, meaning { StructField } can allow us to build up a dot seperated name, exactly what we need to indentify a field within a struct. This is placed in the Primary Rule as well as a few other spots.

Remarks:

As mentioned our main downfalls was the interactions between arrays and structs. There were three planned features which we came short of implementing. They are creating an array of structs, having fields contain arrays and assinging a struct to the values of another struct.

We attempted to create an array of structs and all the correct space was allocated as the size of structs was relativly straight forward to obtain. The problem was that we referenced each field in an obj by its unqiue name. If we have an array, the fields lose their unique identifier and hence we can not make them objects anymore. We could use some combination of indexes as a part of the identifier but this was a possible deep rabbit hole when you consider we also have to be able to assign to these values which our current ArrayAssignment production rule is no where close to being able to implement.

Our issue with arrays in a struct was not a design fault but rather a difficult to implement feature given the design we have. If we were to implement arrays in the context of a Struct Definition, we have issues with field sizes, offsets, determing which fields are arrays and which are not (this information is stored in the object itself, not the field). An attempt was made to expand our Struct Def and ATG to handle arrays in a struct using our production rule ArrayDecl (essentially returns an array of dimensions) but this lead to more and more

issues. We decided for the sake of simplicity and time to forego this feature. It is understood that this feature is highly desired as it allows structs to better represent real concepts. Perhaps given more time, our arrays could be made more flexible to fix both issues present but they are not at this time as the multidimensional factor of our arrays is complex in its own right.

Our last issue we had with structs was with assigning a struct object to the values of another struct object. This issue arose out of attempting to have structs act as Types. We stored structs as Obj's with the correct Type. This allowed us to easily integrate Structs into the existing framework, and treat statements such as Foo.bar.myInt as a Primary Expression. This has its benefits in terms of simplicity but we face an issue in that Primary will return a Type which we need but Primary will load only a single value taken from the stack into a reg. We could however, attempt to load an address into that register and copy the block of values that represent the struct object into our destination set of addresses. The problem here is seperating out rules for StructFields and ArrayIndexes following an Ident in primary. This is solvable but regretfully we only speculate at a solution here as our attempted solution did not work and we did not want to upset the delicate balance the ATG was in at this point. An easier solution to this problem would be to treat everything (including primative) as an object, even if as a psuedo object. This would allow Primary to return non single register values and consider our Struct Types as a Primary member of the language.

Learnt:

There is a trade up between simplicity and functionality. We felt we emphasised ATG simplicity too much and instead attempted to carry most of the heavy lifting towards the Symbol table. While we do not think this is a bad approach, the symbol table would have to be designed with these ideas attempted in mind. We attempted to change the Symbol table to suit our needs but this was mostly adding on to a very chaotic symbol table as is and a more extensable solution would have been ideal.

We did learn however that you can create complete compile time definitions of programming structures (i.e. struct definitions, class definitions) with relative ease simply using the Symbol Table. We would perhaps consider in the future if we were to create our own compiler to focus primarily on the features of a Symbol Table in an extensible way before implementing language constructs.

We also learnt that our design of arrays was perhaps flawed. This in conjunction with our over ambitious desire to treat Structs with the same degree of functionality as a primitive single valued Type caused many issues. As previously stated in the remarks, treating each member of an array as an Object opens many new avenues for modelling arrays and perhaps that would be a better approach. (We could define a new ArrayObj in the symbol table which could hold references to each Object it contains for example)

Runtime array bounds checking

Note: This was implemented previously when arrays were implemeneted. This will mostly be a rehash of what was covered there when discussing arrays. We were told this is satisfactory as part of the criteria of completing this assignment,

Specification:

A runtime method for checking if a specified array bounds is outside the limits of an array's actual bounds.

Method:

To implement runtime bounds checking we need two bits of information, the bounds of the array and the specified index. Our required equation is simply to make sure:

$$Index_N < Array\ bound_N$$

Where N = dimension of the array.

Our array bounds can first be gotten by looking at our symbol table definition of an array.

```
//for arrays
public int[] dimensions; //dimensions of the array
public int sizeRequired;
```

As we can see it has a very simple definition, it is an object with an int[] to specify its dimensions where dimensions[N] refers to the bounds of the array in dimension N.

i.e. Let dimensions = { 4, 5, 6, 8} , the upper bounds of the array for dimension 3 is 8 -> dimensions[3] = 8. We also know that the arrays dimension count is equal to dimensions.Size .

The second piece of infromation is the index, this is calculated at runtime and is stored in a certain register, the result of an Expr. Knowing this, we can load in the value of a dimension and check that against the value generated by Expr.

```
Console.WriteLine(";Bound check begin");
gen.MoveRegister(compareReg, exprReg);
gen.LoadConstant(boundReg, 0);
gen.RelOp(Op.GEQ, compareReg, boundReg);
gen.BranchFalse(l2);

gen.LoadConstant(boundReg, arr.dimensions[dCount++]);
gen.RelOp(Op.LSS, compareReg, boundReg);
gen.BranchFalse(l2);

gen.BranchTrue(l1);
```

As we can see, we load in a constant 0 (our index must be >= 0) and compare to the compareReg which holds our index value. We then also load in our current dimension we are checking using *arr.dimensions[dCount++]*. We perform the same logic except making sure it is strictly less than our dimension bound. The '++' is simply so we can reiterate through this for each bounds, where dCount specifies which dimension we are currently checking against.

Our last check is to make sure all dimensions are specified as we do not allow partial array indexing in our implementation. This is performed by a simple check at the end:

```
if(dCount != arr.dimensions.Length)
  SemErr("Must specifiy all " + arr.dimensions.Length
    + " indices in array \'" + arr.name + '\'');
else
  gen.MoveRegister(reg, accReg);
```

We give a compile time error if not all indexes were specified.

i.e. int myArr[3][4][5][6][7]; myArr[0][0][0][0][0] := myArr[1]
would give us a compile time error as we would have to specify all 5 dimensions

Our last point is how do we provide a runtime error? We chose to output a string indicating that the specified index was out of bounds and then stop the program from running. We did not see any other alternative and stopping the program seemed appropriate as it is undefiend behaviour if we allowed it to continue.

```
gen.Label(l2);
gen.WriteString("index must be between 0 & ");
gen.WriteInteger(boundReg, true);
gen.StopProgram(tab.programName);
```

Remarks:

This was all implemented in our previous assignment to implement arrays. There is not many remarks on our runtime array bounds checking as we did not see many choices present themselves in terms of how to implement this feature.

The one difference we could have implemented is first checking if the value for the index is a ConstExpr (A compile time known value). This would allow us to perform some kind of array bounds checking at compile time. It was however thought to be simpler to treat all specified indices as runtime Expr, even if we do know its value at compile time.

Learnt:

The main principle learnt when implementing this feature was that the more information you can encode into the symbol table during compile time, the simpler the runtime code can be. Performing run time tasks is a much more difficult then compile time.