CS3071 Final Report
Seán Durban (14320715)

This is the report is the combination of all reports submitted for exercises 3-7.

The aim of **3rd exercise** was to add to SymTab in order to generate comments for each identifier. I first had a look at all the files involved in the exercise (TastierProgram.TAS, SymTab.cs and Tastier.s). I noticed that the original taste program had no comments but when the program was compiled the assembly file did have comments for most of the instructions. I searched the other files and found that the CodeGen.cs contained the code for generating these comments based on the instruction. From CodeGen.cs I could see how to add the comments to the assembly file. In SymTab, I figured that the top scope object would contain all the identifiers in itself in locals. Locals took the form of a linked list with the top scope as the head.

After these discoveries I got to writing the code. I first printed out the integer values of the objects attributes and then when I was sure it was compiling correctly I changed it to output Strings. I declared arrays of strings where the index corresponded to the int value of the objects attribute, I decided on them being constants so they can be used again if necessary in future modifications. As procedures are always of type undef, I also changed the method so it would only output its name and kind when it was a procedure.

The aim of the **4th exercise** was to extend the current Tastier language to allow the use of constant integers and booleans.

Constants in programming languages are usually global values that can not be altered during the execution of the program. Also constants are preceded by a keyword ‘const’, ‘final’ etc. I first looked at the base tastier code provided, I was particularly interested in the Tastier part (of the ATG file) where it looked for the keyword ‘program’, also the VarDecl where it recognises and creates the new variable. Then how the object was stored in Stat.

First I changed the object variable in SymTab so that it could be a constant. I did this by creating another attribute of Obj called ‘constant’, I could have added a new kind but decided on this method for clarity. As obj had a new attribute then I had to augment to the constructor to update the constant attribute accordingly. I in turn had to change all the obj constructor call in the .ATG file to reflect this.

Next I added the ‘ConstDecl’ to the .ATG file so the compiler would be able to correctly assign a constant value. I decided to assign ‘Final’ to act as the keyword to describe the start of a constant declaration. I also took the decision to only allow a single constant value to be declared at once. I accordingly adjusted other parts of the .ATG file to look for both constDecl and varDecl. As this exercise was to extend the current code, I kept the current initialising method of declaring the variable then assigning it a value.

I then had to update the stat method of the .ATG file to ensure that constants were only assigned a value once. I did this by including another attribute to the obj in the SymTab, it's called 'assigned' this value is incremented every time the value is assigned a value and stored in memory. So if the obj is a constant and it hasn't been assigned a value yet then it is assigned and stored, otherwise it is not and an error is raised.

The aim of the **5th exercise** was to extend the context free grammar for Tastier so that the language provides for the use of one dimensional arrays, and modify the Tastier.ATG and SymTab.cs files to reflect your changes.

First I had to figure out what the syntax for an array was. In most programing languages an array is declared with its type, name and square brackets ('[' & ']'). I decided on the following syntax for my arrays declaration:

Array <type> arrName[<size>];

I chose to use the keyword 'Array' to signify the start of an array declaration as it is easier to deal with as the designer of the language as without the keyword the declaration would be close to the declaration of a variable. The size must of a constant int value, it allows the memory for all the arrays elements to be allocated.

I also had to figure out what an array is and what it can do. I came up with the following uses:

- myArr[<Expr>]=<value>;
- myArr = {<X values>}; where X is the size of the arr.
- myVar = myArr[<Expr>]; where Expr could be a constant or variable of type int.

One of the main characteristics of an array is that it is a contiguous data structure. This means that I needed to allocate the space for all elements of the array when it was declared. The Array object will have an adr and then all the elements can be accessed at the adr+(index*elemSize). To figure this out I drew the following memory diagram:

Value	*OtherMemory*	myArr[0]	myArr[1]	myArr[2]	myArr[3]	myArr[4]	*OtherMemory
Mem Adr		adr+0(0*4)	adr+4(1*4)	adr+8(4*4)	adr+12(3*4)	adr+16(4*4)	nextAdr ptr

Building on this diagram I looked at the symtab to see how objects are currently assigned an adr. I found that the scope had a 'nextAdr' variable which was the value of the next free adr and was incremented every time a new object was created. So to allocate the space I got the scope that the array is declared in and then I add the size of the array to that scope's next adr. This ensures that the space for all the elems is left clear by future objects being declared, as seen in the above diagram the nextAdr pointer is moved to the memory space beyond/after the last element of the array.

Next was the assignment of the array elements, I decided to allow for both the assignment types noted above. The first was the individual element assignments, code was added to the Stat for the following syntax:

Ident [SimExpr] := Expr ;

The Expr value is then stored at the adr of the arr elem adr[SimExpr]

Another form of assignment is when a whole array is assigned at once. The following code was added to Stat to to deal with this:

Ident := {SimExpr, SimExpr, ….};

The code checks that the correct amount of elements are assigned in this statement. A variable ‘n’ is used to keep track of the current index. Every time a SimExpr is stored in memory it is incremented. I used the code gen to get a register and load current value of n in that register to be used in the storage functions.

One Issue I encountered was that when generating registers to use when storing the elements (instructions to load the address, add the offset etc) were using registers which the increment of the register used for the previous element. In order words R6 and R7 would be used for arr[0] then R8 and R9 used for arr[1]. This was a clear problem as the program would run out of available registers quite quickly. To address this issue I cleared the registers after the element was stored as their values were then redundant.

Next use to cover was a variable being assigned an array element. I added code to the Primary part of ATG, this is used to load the value of the element at the array index given into a register and return that register. Then the value in that register is stored in the adr of the var as usual.

In Symtab I added an obj attribute called ‘isArray’ to show whether an obj is an array or not. This is used in the atg for array only parts. Also ‘arraySize’ is used mostly for error checking and is assigned when an array is Declared. I plan on changing this implementation so that an array is a obj kind. This is make more sense and improve readability of the code.

The aim of the **6th exercise** was to Extend the programming language described by the attributed translation grammar for Tastier to include a conditional assignment statement and some form of structured loop statement.

Both parts of the assignment would depend on branching which I had not used previously in Tastier. Therefore I studied the implementations of both the if statement and the while loop. I also inspected the CodeGen to see how the expressions were compared and how the branching labels were written.

After I the formed the syntax of the conditional statement:

<identifier>:=<condition>?<expression$_1$>:<expression$_2$>;

I then wrote out how a conditional statement would look in ARM so I would know the amount and the position of labels and branches necessary.

```
        CMP Rx, Ry
        B__ L1
        Expr1
        B L2
L1
        Expr2
L2
```

Using both of the above I went about adding to Stat in the ATG file to allow for the conditional statement. The grammar is as follows:

<Ident> ":=" ['('] <Expr$_{Cond}$> [')'] '?' <Expr$_1$ > ':' <Expr$_2$ > ';'

I chose to allow for optional brackets around the condition out of personal preference, as I usually surround it in brackets when programming for readability. Another design choice was to just load the result to be stored based on whether the condition was true or false into a temp register and then store the result in that temp register into memory at the end of the statement. This meant that I could avoid the duplicated code I had in a previous iteration to store the value. Before the compiler was generating code to store the result for both expressions when only one would be run by the program per execution.

Next was the for loop and I started with the syntax:

for (initial action; update action; terminating condition)

As before I went about writing the ARM code out to understand the branching.

```
        ;initial action
        B L2
L1
        ;update action
L2
        CMP Rx, Ry
        B__ L3
        ;code in for loop
        B L1
L3
```

The branch to L2 is to avoid the update action on the first run. The initial action is only run once therefore is outside the loop too. The program will run the code and branch back to L1 until the condition is true and then it will branch out to L3 which ends the loop.

Using both of the above I went about adding to Stat in the ATG file to allow for the conditional statement. The grammar is as follows:

"for" '(' Stat Stat <Expr$_{Cond}$> ')' Stat

There's no need to explicitly look for ';' and '{' '}' as Stat already takes care of this. Also Stat can be 1 or more Stat(s). I chose to use this syntax for the for loop compared to that of another programming language such as java, as I think the order makes more sense when programming and would be my personal preference of the two. It would also be less straightforward.

The aim **7th and final exercise** was to modify the context free grammar so that parameters can be passed by reference in procedure calls, and add one extra feature. The extra feature I chose to implement was post increment and decrement (i++, i--).

Note: Even though you requested that we do not modify the CodeGen file, in this exercise it was required for getting the Global address of an object. I implemented the changes as suggested by a student (Dario) and shared with the class, we had no confirmation before the deadline of the CodeGen was intentionally like this. But the changes made the GlobalAddress function work as intended and the same as the LocalAddress function.

Passing parameters by reference means that the parameters should act like pointers to the object being passed.i.e If a function has a single parameter 'a', it should contain the address of the object the caller passes for a. It has the benefit that the function can change the value of this object for uses outside the current scope too. So value of addr at 'a' would change in the function called and the caller.

The first step was to modify the ProcDecl to allow for parameters to be declared with the function declaration. To do this I created a ParamDecl, this is essentially just the ("int a") part it gets a type and an ident then creates an object with the same type and ident in the functions scope. Therefore that ident can be used in the function as a variable. ParamDecl outputs the parameter's name (ident) which is added to an array for later use.

Next I had to come up with a way to hold or store the references between the caller function and the function being called. After brainstorming and trying a few different ideas I settled on a straight forward approach of using registers to hold the references. All registers are deemed free at the point of calling another function therefore the first parameter is stored in R5, the second in R6 and so on. This has a clear limitation in that it will run out of registers quite quickly and some register (TOP) should not be corrupted. But it will work well for use in my tastier language where there's no functions with a high amount of parameters. A check is added to ProcDecl to ensure the number of parameters isn't above 6 (R5-R9,R12).

Now that we know the registers will contain the references to the parameters we must add to ProcDecl to store the references in the parameters declared in this function. The array mentioned earlier contains all the idents of the parameters. Therefore after the labels for the

function have been generated we can load the references from the parameter registers. For every parameter declared, store the reference (addr of obj).

Also we would have to move the references into the registers before calling the function. This part in in the Stat. For every parameter, get its object and load it's address. Then move this address to a free register (R5 onwards). I have included a check so that registers R10(BP) and R11(TOP) cannot be corrupted.

As the parameters would just contain an address I had to add to every Load and Store to check if the object was a parameter. I added a boolean attribute to the objects in Symtab for this. If the object is a parameter then its address is loaded into a register and then moved into R2. Then the value is loaded/stored using the address in R2. This allows for the value to change in the caller scope too.

Another limitation to the parameters is that it can only handle ident when calling the function, not constants. As the parameter are passed by reference, which a constant would not have as it has no address.

I chose the post increment and decrement as my extra feature as it is something I use a lot when programming and feel it is particularly useful for loop, which were done in a previous exercise. I implemented it in the form: i++; and i--;

I added to Stat so that after an Ident it could be either "++;" or "--". For both I load the value into a register then load the constant 1 into another register. To Apply the operation (+ or -) I then call AddOp in the CodeGen. After the operation is applied then the result is stored back.