

Please note: Had to add 'using System.Collections' to Parser as I am using ArrayList in ATG, also added few function in CodeGen for parameter passing.

### **Constant Report**

To define a constant I made a new symbol "<-" to distinguish from variable declaration ":=".

constant keyword was also added to set apart constants and variable. Also in the .TAS file, constants can be declared locally or globally.

Since we cannot redefine a constant, and try to redefine, it will give us a compilation error, since it throws an error, and in the tastier.s file it gives "cannot redefine a constant variable compilation error(s)".

When referring to a constant variable, the keyword 'constant' should be mentioned. For example

```
total := total + 'constant' X
```

this tells the compiler that X is a constant

### **1-Dimensional Array Report**

For this assignment I extended the context free grammar for Tastier for the use of 1D array. My 1D array implementation currently only supports int and bool types. The following is the syntax to declare an array:

```
"array" type identifier "[" number "]" ;
```

The array keyword must be used in order to declare an array, or access an array. The array can be declared locally or globally. The ArrayDecl in the Tastier.ATG file expects the "array" keyword followed by type and identifier, followed by "[" and a number followed by "]". The string value is converted into an int using `Convert.ToInt32(t.val)`, this value is the length of the array. When the identifier is encountered, an object by its name (if an object by that name has not already been created ) is created and the length of the array is stored in the symbol table. In the SymTab.cs, I have different approaches to allocate memory space for different objects, depending on either if the object is a scalar or non scalar. An array is considered non scalar and with the aid of the array's length I am able to allocate required memory space on the stack for

the array in the SymTab.cs.

The following is the syntax for array statements: “array” identifier “[” index “]”  
“:=” Expr “;” array x[3] := 7;

In the Stat section, for array statements, it finds the array object and before any other action is taken, it checks if the index defined is within the bounds of the specified array, if it isn't, an “array out of bounds” SemErr is thrown.

If index is within the bounds, the next step taken is to store the value which is in the register, reg of “Expr” into the array index. Hence, the function StoreIndexedGlobal or StoreIndexedLocal is called from the CodeGen file depending on the lexic level.

If a variable wants to access the value of a certain array index, the syntax is as following:

```
int var; var := “array” x[“index “];
```

Firstly for this implementation, I check whether the array specified is actually an array, if it is not a SemErr is thrown informing that it expects an array. The second step I take is to ensure that the index specified is within the array, if it is not then SemErr is thrown saying “array out of bounds”. To transfer the value stored in a particular array index into a

```
mono tcc.exe TastierProgram.TAS > Tastier.s
-- line 27 col 15: array out of bounds
make: *** [compile] Error 1
```

variable, the LoadIndexedLocal or LoadIndexedGlobal functions are called from the CodeGen file depending on the current lexic level.

An improvement I could have made for my 1D array implementation was that instead of using number when declaring an array, instead of using number, I could have used an expression.

## Conditional statement and for loop statement Report

The aim for this assignment was to extend attribute translation grammar for the Tastier language, so it supports conditional assignment statement and a structured loop statement.

### Implementation of conditional assignment

The following is the syntax for a conditional assignment:

```
<identifier>:=<condition>?<expression1>:<expression2>
```

Here is the code snippet from .TAS file

```
t1:= 3;
t2:= 10;
t3:= t2?t1?t2+4:t1-2;
```

To implement the conditional assignment, I had to extend the Stat section in the ATG file.

a Statement could be an identifier followed by a “:=”, followed by an expression. This expression may also be followed by a ?, this is achieved by following the Expr with [] which tells the compiler the grammar after the [ is optional. The ? acts as a guard/condition, if the expression evaluates to True then the expression before “:” is the value of the identifier else the expression after the “:” is the value of the identifier.

In the above snippet, t3’s value is either t2+4 or t1-2 depending on the evaluation of t2>t1.

Once the Expr is evaluated, if the expression evaluates to False then we branch to the second expression, this is done by executing gen.BranchFalse(L2), the label L2 is generated just before the “:”. If the original expression (the expression to be evaluated) is True, then we simply follow through and the identifier gets the value of the first expression. After we store the result of the expression into identifier, we branch to end, so we don’t execute the code for the second expression.

14318618

If the first expression evaluates to true, we should exit the conditional statement and not evaluate the next expression or else an incorrect result will be obtained.

### Implementation of the for loop statement

The following is the syntax for my implementation of Loop Statement:

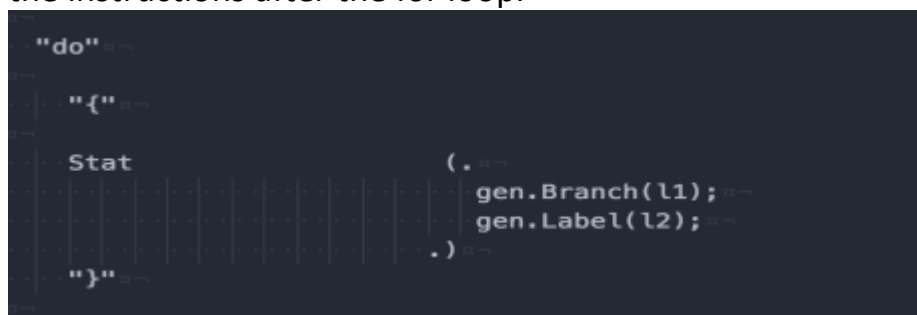
```
for (initial action; update action; terminating condition)
  do {    Stat}
```

And this is the code snippet from TastierProgram.TAS for a for loop

```
r:=0;
for(ix:=1;ix:=ix+1;ix<3;)
do
{
r:=r + 1;
}
```

The concept behind the implementation of the for loop structure is similar to the conditional statement structure grammar, at each iteration we evaluate the terminating condition. If it evaluates to True then we execute whatever instructions are inside the do. After we execute the instructions inside the do, we loop/branch back to the same terminating condition and re-evaluate it, if this time the condition evaluates to False we exit the for loop, and branch out and execute instructions after the for loop.

Below is a code snippet of the do {...} statement, the Stat is executed if and only if terminating condition is True, and we branch back to label L1 which is where the terminating condition is. When terminating condition evaluates to False we branch to the Label L2, so Stat is not executed and we continue with the instructions after the for loop.



```
"do"
  "{"
  Stat
  (
    gen.Branch(l1);
    gen.Label(l2);
  )
  "}"
```

## Exercise 7

### Implementation of switch statement

For my extra feature I implemented switch case statement. Unlike if else statements, the switch statement can have a number of possible paths.

The syntax for the switch statement is:

```
switch(variable){  
  
case 1: Stat; break;  
case 2: Stat; break;  
case n: Stat; break;  
default: Stat; break;  
}
```

The Stat and break being optional.

The switch expects an Ident; once it finds the object in the symbol table it loads the value of the variable into a register, for this purpose let us call the register reg.

The value of reg is then compared with the cases, each case expects an Expr<reg2,type2>. The value of the register to be compared is stored in reg2, hence both reg(variable to be compared) and reg2(case value) are compared by calling gen.RelOp(Op.EQU,reg2,reg). If the comparison is False, then we branch to the next case else we exit the switch case statement. If none of the case values match the variable's value then it will fall through and execute the default statement and then exit the switch case statement.

## Passing Parameters

When the compiler sees a variable declaration, it sets up a memory location for each variable that is declared. The compiler keeps a symbol table which keeps track of the relationship between variable's symbolic name and the address of the memory location where the variable will reside.

```
int i,j;
```

```
float x;
```

```
char c1;
```

(In this example I am assuming int take 2 bytes, float 4 bytes)

This is how the symbol table will look like:

variable	address
i	2000
j	2002
x	2004
c1	2008

So as the variable are set up in memory this is how memory will look like:

2000	space for i
2002	space for j
2004	space for x
2008	space for c1
2009	

## Passing parameters by Reference

When the compiler sees a subroutine call that has one or more reference parameters, it generates machine code to copy the address of the actual parameter to the stack. Since all the address calculations are done in R2, the method, `moveRegister` from `codeGen` was called to move the address to a fresh register which is obtained by calling `gen.getRegister()`. The address then is pushed onto the stack by calling `gen.pushParam(adr)`. This pushes the address onto the system stack.

Before entering a procedure call we first branch to the enter subroutine and do the necessary housekeeping to enter the procedure, we push the parameters, the return address, lexic level delta, static link and dynamic link and then we reserve space for the local variables, respectively.

At the end of every procedure, the compiler will generate an instruction that will reset the state of the stack as it was before calling the procedure. Because by definition the stack pointer, `TOP`, will point at the topmost value on the stack, changing its value will effectively “delete” the parameters. The stack pointer register holds the address of the topmost value of the stack at runtime. The space allocated for parameters vary from one procedure call to another depending on the number of parameters and their types.

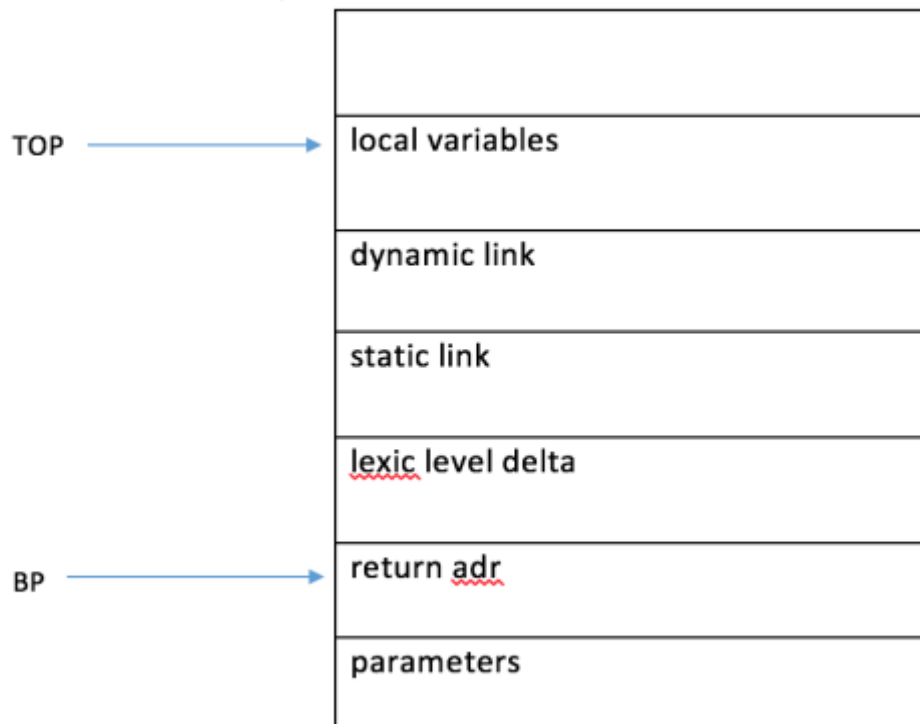
As the parameters are pushed onto the stack, an offset for each parameter is recorded relative to the base pointer. For my implementation, the parameters are stored in an `ArrayList` and the position of each parameter is the offset.

the syntax for passing parameters is as following:

```
foo(x;y;)
```

Each parameter must be followed by a ‘;’

This is the stack frame layout:



suppose if the function `foo(num1, num2)` is called; the compiler would issue machine language instructions to copy the address for `num1, num2` order onto the system stack (In my implementation the parameters are pushed as they appear, First Come First Serve order). When the code executes, the stack will look similar to this:

(this drawing assumes address takes 4 bytes)

4000	<u>&amp;num2</u> 2002
4004	<u>&amp;num1</u> 2000
4008	~

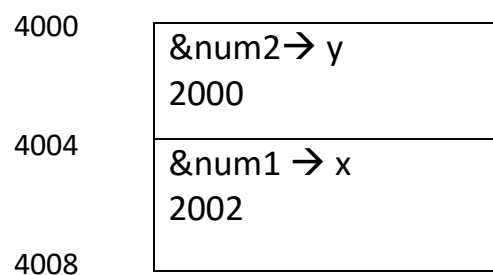
The offsets of these parameters are linked to the procedure object in the symbol table, as mentioned earlier the parameters are pushed as they are encountered, since `num1` is seen first, it will be pushed first and `num2` will be the last parameter to be pushed. The parameters are added to the `ArrayList` and reversed at the end, so the index of the parameters match the offset in the stack. For example `num1` is added to the `ArrayList` and then `num2` is added. This `ArrayList` is then reversed so `num1` is at position 2 and `num2` is at



position1(this is not the case at least in C, as they are pushed in the reverse order).

Suppose if the subroutine foo looks like this:

```
void foo(int x,int y){
total := total + x;
total := total + y;
}
```



When the procedure refers to one of its parameters, the compiler generates an instruction that adds the subtracts the BP and offset in the symbol table. This will give the address in the stack that holds the address of the variable. The instruction then fetches the variable address from this location. Then the compiler proceeds to retrieve the value of the variable at the address fetched from the stack.

For example to get the value of x when evaluating Statement total:=total + x, the compiler will generate an instruction that will subtract the BP value and the offset of x (which in this case will be 2 as num1 is in the second element in the ArrayList) to give the value 4004. This 'points to' the address of num1, the address is then loaded into a register and another instruction retrieves the value stored at address 2000.

The implementation also has parameter type checking and that correct number of parameters are passed.

For the implementation for both pass by reference and pass by value, I would have concatenated '&' for pass by reference and '\*' for pass by value to distinguish what to push onto the stack(the address of the variable or the value of the variable).