CS3071 Compiler Design

Shane Fitzpatrick 09487581

Lab 7

This lab asked us to extend the programming language described by the attributed translation grammar for Taste to include a conditional assignment statement of the form:

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

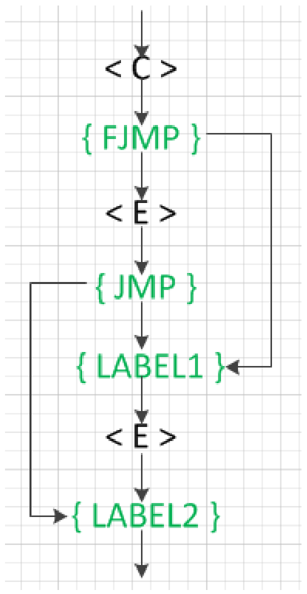
The first alteration I made was changing the assignment operator from "=" to ":=" throughout the ATG and EBNF. Next, I changed the 'Stat' production to include the conditional statement described in the lab.

< Stat > -> < Expr > '?' < Expr > ':' < Expr >

There was already support for assigning an identifier to an expression in the ATG, so all I had to add was an optional conditional part, i.e.

```
Stat = ident
        ( ":=" Expr [ "?" Expr ":" Expr ]
        …
```

The flow diagram that I drew up for this lab was:



No changes were necessary to the SymTab.cs or CodeGen.cs files.

<u>Lab 8</u>

The requirements for this lab were to add support for constant definitions, the three missing relational operators “!=” (not equal), “<=” and “>=”, string data types and to modify the “write” statement to handle multiple values on a single line.

Alterations had to be made to the EBNF, ATG, SymTab.cs and CodeGen.cs files to add support for constant definitions. I added the following production to the EBNF and ATG.

< ConstDecl > -> “const” ident “ :=” < Expr > < ConstList > “;”
< ConstList > -> “, “ ident “:=” < Expr > <ConstList >
< ConstList > -> ε

The production took the following form in the EBNF:

ConstDecl = “const” ident “ :=” Expr { “, “ ConstDecl } “;”

In the ATG file, I added a constant object kind to distinguish constants from variables and procedures. I also altered some code in the ‘Factor’ production to support my new constant kind. The ‘ConstDecl’ production described above was also added to the ‘Taste’ production, to allow global constant definitions, and to the ‘ProcDecl’ production, to all local constant definitions. Constant values get assigned when they are declared, and a SemErr is output if there is an attempt to assign a value to a constant elsewhere.

In the SymTab.cs file, I added a constant kind and altered the ‘NewObj’ method to provide constants with the correct address value.

In the CodeGen.cs file, I added another for loop in the ‘Interpret’ method. The function of this loop is to allow the assignment of global variables before the ‘main’ method. I also added support for the Op codes I felt were appropriate.

Alterations had to be made to the ATG and CodeGen.cs files to add support for the missing relational operators. The ‘RealOp’ production was changed to include these operators, so it became:

RelOp = "==" | "<" | ">" | "!=" | "<=" | ">="

In the CodeGen.cs file, I added new Op codes to handle the new operators. I fashioned these on the existing Op codes for the operators that there was support for.

For string data type support, I made alterations to the ATG, SymTab and CodeGen files. In the ATG, I added a ‘str’ type to distinguish string data types from integer and boolean. I then added this type to the ‘Type’ production to recognize “string” as type ‘str’. Next, I added ‘otherchars’ to ,so strings could support every character on my keyboard, and dQuote to ‘CHARACTERS’ and added a ‘strT’ Token. The ‘strT’ token takes the format:

strT = dQuote { letter | digit | otherChars } dQuote.

I then altered the ‘Factor’ production to include my new ‘strT’ token. The final alteration made to the ATG was to allow the ‘write’ statement to write string values.

The only alteration I made to the SymTab.cs file was to add a ‘str’ type to distinguish the string data type from integer and boolean.

In the CodeGen.cs file, I added two new Op codes for strings, STR and WRITES, new methods to use with these Op codes and string array (strStack) to hold the string values. The Op code STR is used to push the index of the string in the ‘strStack’ onto the ‘stack’ that was already implemented. Next the code gets the length of the string and uses it to create the correct string of that amount of characters and pushes the result onto the ‘strStack’. This is achieved using my new methods NextS() and PushS().

The new Emit() for strings takes an Op code, string value and integer length as parameters. The method pushes the Op code and length to the code stack. Next, the string value is converted to a byte[] and each byte is added to the code stack. The NextS() method takes the length of the string as its sole parameter. It then creates a byte[] of that length and fills it with the bytes of the characters from the code[]. Finally, it converts the byte[] to a string and returns it. The PushS() method takes a string as a parameter and adds it to the ‘strStack’. The index of the string stack is stored in an integer variable called sIndex.

The Op code WRITES gets the value from the strStack and writes it through the console. I choose to keep the quotes as part of the string datatype so they are stored with the string and written to the console.

To extend the functionality of the write statement to include the ability to output multiple values on a single line I did the following. In the ATG I changed the write statement to only write integer or string values and to support multiple Expr on the same line.

< Stat > -> “write” < Expr > < WriteList > “;”
< WriteList > -> “, “ < Expr > < WriteList >
< WriteList > ->   ε

The production took the following form in the EBNF:

```
Stat = …..
         | "write" Expr {", " Expr} ";"
         …...
```

In the CodeGen.cs file, I added the Op codes WRITEC and WRITEL, and changed the Op code WRITE. My new definition for WRITE has similar functionality to that old apart from instead of writing a line, it just simply writes the value. WRITEC writes a comma and a space (“, “) and WRITEL writes an empty line. The ATG uses these Op codes by writing a value, followed by “, “ if necessary and more values, and finally writes a new line.

Lab 9

The requirement for this lab was to extend the programming language described by the attributed translation grammar for Taste to allow for the use of simple ‘for’ loops (where the initial and final values for the index variable are numeric constants) of the form:

for i := 1 to n do statement

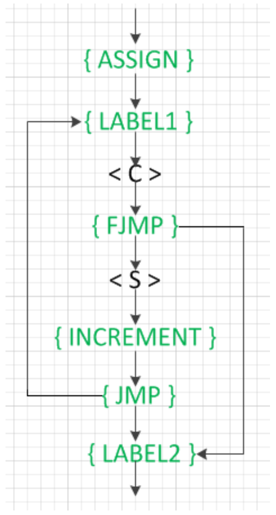
I altered the ‘stat’ production to allow for this change:

< stat > -> “for” ident “:=” const “to” const “do” < stat >

This took the following form in my EBNF:

Stat = …
| "for" ident ":=" (number | ident) "to" (number | ident) "do" Stat.
….

I also added support for using variable identifiers as the initial and final values in the ATG. No changes were necessary to the SymTab.cs or CodeGen.cs files. Here is my flow diagram for this statement:



Lab 10

The requirement of this lab was to extend the programming language described by the attributed translation grammar for Taste to allow for the declaration and use of one-dimensional arrays of integer values. The indices to the elements of an array of n values should range from

1 to n. I made alterations to the ATG, SymTab.cs and CodeGen.cs files to achieve this.

In the ATG, I changed the VarDecl, Stat and Factor productions to support arrays. These productions now take the following form in the EBNF:

```
VarDecl =    Type ident [ '[' number ']' ] {"," ident [ '[' number ']' ]} ";".

Stat =   ident ( ( '[' number ']' | )
         ":=" Expr ["?" Expr ":" Expr ]
                 |"(" ")"
                 ) ";"
         |
         ….

Factor =     ( "(" Expr ")" | ident( '[' number ']' |) | "true" | "false" | number | "-" Factor).
```

To distinguish array from non-array variables in the SymTab.cs file, I added a boolean attribute to Obj - isArray. When the NewObj() method is called, I assume it is not an array. I then added a new method, toArray(), to convert an object to an array. The method will set the ‘isArray’ and ‘size’ (new attribute to hold the size of the array) of the Obj, and alter its address to take into account the additional scope.

In the CodeGen.cs file, I added two new Op codes, STOA and STOGA, to store local and global arrays. These Op codes get the address and size of the array from the stack and initialize the correct number of places on the stack to zero. When arrays are being indexed in the ATG, the value of the index is added to the address of the array to get the correct value at that index.

Lab 11

For this lab I chose to add support for char data types, a ‘do-while’ statement and run-time checking of array bounds.

For the char data types, I altered the ATG to include a ‘sQuote’ in CHARACTERS, a ‘charT’ token and added the token to the ‘Factor’ production. The token took the following format:

```
charT =  sQuote  ( letter  |  digit  |  otherChars )  sQuote
```

No alterations needed to be made to the SymTab.cs or CodeGen.cs files because I implement the characters as strings of length 3 (the single char and two single quotes). This results in them having all the same functionality as string data types.

My ‘do-while’ statement takes the form:

```
do  { statement  }  while  ( condition );
```

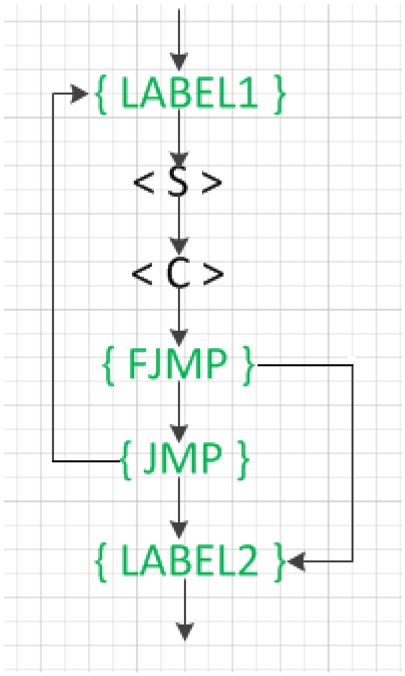
I altered the 'stat' production in the ATG to allow for this statement:

< stat > -> "do" < stat > "while" < Expr > ";"

And in the EBNF:

Stat = ….
| "do" Stat "while" "(" Expr ");"
…..

My flow diagram for this statement:



The run-time checking of array bounds was easy to implement in the ATG alongside my array code. An SemErr is output if the program initialises an array with size less than 1. Also, when an array is indexed the index is checked to see if it is in bounds. If not, then a SemErr is output.