Open access Journal International Journal of Emerging Trends in Science and Technology

Impact Factor: 2.838 DOI: http://dx.doi.org/10.18535/ijetst/v3i06.05

Etensible Compiler-"STAT"

Authors

Sushama[1], Mrs Reema Sachdeva[2]

[1]M-Tech student of Dept of Computer, Science and Engg, Sat Kabir Institute of Technology and Management Bahadurgarh, Haryana-124507

Email: *Sushama.angel@gmail.com*

[2]A.P. in CSE Dept, Sat Kabir Institute of Technology and Management Ladrawan, Bahadurgarh Haryana-124507

Email: *arorareema@live.com*

ABSTRACT

A compiler is a computer program or a set of program which converts the data from source code to object code, source code mean human understandable form whereas object means machine understandable form i.e.binary language. The compilers made till know are used to transform the a specific language and provides only the features which are added in it while designing the compilers, no other features are supported by the compiler other than those mentioned while designing it. This problem can be solved using extensible compilers. The basic idea used here is to extend a programming language by by adding new syntax, features etc. through adding extension modules which act as plug-ins for the compiler. Certain challenges are faced while building such compiler like creation of extensible that are simultaneously powerful, to allow effective extensions, convenient to make these extensions easy to write; and composable, to make it possible to use independently-written extensions together.

INTRODUCING "STAT"

I have made an extensible compiler and named it as "STAT".

So in the development that follows, I'll actually be doing a top-down development of BOTH the language and its compiler. The BNF description will grow along with the compiler.

GETTING STARTED

Given the BNF above, write a parser that just recognizes the brackets:

```
{Parse and Translate a Program}
Procedure Prog;
begin
Match (' p ') ;
Header ;
Prolog ;
Match (' . ' ) ;
Epi log;
end;`
```

The procedure Header just emits the startup code required by the assembler:

```
{Write Header Info}
procedure Header ;
begin
WriteLn ('WARMST', TAB, 'EQU $A01E ') ;
end ;
```

The procedures Prolog and Epilog emit the code for identifying the main program, and for returning to the OS:

```
{ Wr i te the Prolog }
procedure Prolog ;
begin
PostLabel ('MAIN ') ;
end;
{Write the Epi log }
procedure Epi log ;
begin
EmitLn ( 'DC WARMST' ) ;
EmitLn ( 'END MAIN ' ) ;
```

```
end ;
```
The main program just calls Prog, and then looks for a clean ending:
```
{ Main Program }
begin
Init ;
Prog ;
i f Look <> CR then Abort ( ' Unexpected data after '' . '' ' ) ;
end .
```
At this point, STAT will accept only one input "program," the null program:
PROGRAM . (or 'p.' in our shorthand.)

DECLARATIONS

The next step is to process the code for the main program.To parse this definition of a main block, change procedure Prog to read:
```
{ Parse and Translate a Program }
procedure Prog ;
begin
Match ( ' p ' ) ;
Header ;
Main ;
Match ( ' . ' ) ;
end ;
```
and add the new procedure:
```
{ Parse and Translate a Main Program }
procedure Main ;
begin
Match ( ' b ' ) ;
Prolog ;
Match ( ' e ' ) ;
Epi log ;
end ;
```
Now, the only legal program is:
PROGRAM BEGIN END . (or 'pbe.')
The next step is to decide what we mean by a declaration.

DECLARATIONS AND SYMBOLS

<data declaration> ::= VAR <var-list>
Note that since there is only one variable type, there is no need to declare the type.The procedure Prog becomes:
```
{ Parse and Translate a Program }
procedure Prog ;
begin
Match ( ' p ' ) ;
Header ;
TopDecls ;
Main ;
Match ( ' . ' ) ;
end ;
```
Now, add the two new procedures:
```
{ Process a Data Dec larat ion }
procedure Decl ;
begin
Match ( ' v ' ) ;
GetChar ;
end ;
{ Parse and Translate Global Dec larat ions }
procedure TopDecls ;
begin
while Look <> ' b ' do
case Look of
' v ' : Decl ;
else Abort ( ' Unrecognized Keyword ' ' ' + Look + ' ' ' ' ) ;
end ;
end ;
{ Parse and Translate a Data Declaration }
procedure Decl ;
var Name: char ;
begin
```

INITIALIZERS

```
Match ( ' v ' ) ;
Al loc (GetName ) ;
end ;
```
The procedure Alloc just issues a command to the assembler to allocate storage:
```
{ Al locate Storage for a Variable }
procedure Al loc (N: char ) ;
begin
WriteLn (N, ' : ' , TAB, 'DC 0 ' ) ;
end ;
```
Adding this syntax to Decl gives this new version:
```
{ Parse and Translate a Data Declaration }
procedure Decl ;
```

```
var Name: char ;
begin
Match ( ' v ' ) ;
Al loc (GetName ) ;
while Look = ' , ' do begin
GetChar ;
Al loc (GetName ) ;
end ;
end ;
```

Change Alloc as follows:

```
{ Al locate Storage f o r a Var iable }
procedure Al loc (N: char ) ;
begin
Write (N, ' : ' , TAB, 'DC ' ) ;
i f Look = '= ' then begin
Match ( '= ' ) ;
WriteLn (GetNum) ;
end
else
WriteLn ( ' 0 ' ) ;
end ;
```

EXECUTABLE STATEMENTS

Now an almost usable language is made! What's missing is the executable code that must go into the main program. But that code is just assignment statements and control statements ... all stuff we have done before. So it shouldn't take us long to provide for them, as well.

The BNF definition given earlier for the main program included a statement block, which we have so far ignored:

<main> ::= BEGIN <block> END

For now, we can just consider a block to be a series of assignment statements:

< block >::= (Assignment)_

Let's start things off by adding a parser for the block. We'll begin with a stub for the assignment statement:

```
{ Parse and Translate an Assignment Statement }
procedure Assignment ;
begin
GetChar ;
end ;
{ Parse and Translate a Block of Statements }
procedure Block ;
begin
while Look <> ' e ' do
Assignment ;
end ;
```

Modify procedure Main to call Block as shown below:

```
{ Parse and Translate a Main Program }
procedure Main ;
begin
Match ( ' b ' ) ;
Prolog ;
Block ;
Match ( ' e ' ) ;
Epi log ;
end ;
{ Clear the Primary Register }
procedure Clear ;
begin
EmitLn ( 'CLR D0 ' ) ;
end ;
{ Negate the Primary Register }
procedure Negate ;
begin
EmitLn ( 'NEG D0 ' ) ;
end ;
{ Load a Constant Value to Primary Register }
procedure LoadConst ( n : i n t e g e r ) ;
begin
Emit ( 'MOVE # ' ) ;
WriteLn ( n , ' ,D0 ' ) ;
end ;
{ Load a Var iable to Primary Register }
procedure LoadVar (Name: char ) ;
begin
i f not InTable (Name) then Undefined (Name) ;
EmitLn ( 'MOVE ' + Name + ' (PC) ,D0 ' ) ;
end ;
{ Push Primary onto Stack }
procedure Push ;
begin
EmitLn ( 'MOVE D0,□(SP) ' ) ;
end ;
{ Add Top of Stack to Primary }
procedure PopAdd ;
```

```
begin
EmitLn ( 'ADD (SP)+ ,D0 ' ) ;
end ;
{ Subt ract Primary from Top of Stack }
procedure PopSub ;
begin
EmitLn ( 'SUB (SP)+ ,D0 ' ) ;
EmitLn ( 'NEG D0 ' ) ;
end ;
{ Mu l t i p l y Top of Stack by Primary }
procedure PopMul ;
begin
EmitLn ( 'MULS (SP)+ ,D0 ' ) ;
end ;
{ Divide Top of Stack by Primary }
procedure PopDiv ;
begin
EmitLn ( 'MOVE (SP)+ ,D7 ' ) ;
EmitLn ( 'EXT. L D7 ' ) ;
EmitLn ( 'DIVS D0,D7 ' ) ;
EmitLn ( 'MOVE D7,D0 ' ) ;
end ;
{ Store Primary to Var iable }
procedure Store (Name: char ) ;
begin
i f not InTable (Name) then Undefined (Name) ;
EmitLn ( 'LEA ' + Name + ' (PC) ,A0 ' ) ;
EmitLn ( 'MOVE D0 , ( A0) ' )
end ;
```

The error handler Undefined simply calls Abort:

```
{ Report an Undefined I d e n t i f i e r }
procedure Undefined ( n : st r ing ) ;
begin
Abort ( ' Undefined Identifier ' + n ) ;
end ;
```

The BNF for the assignment statement is:

<assignment> ::= <ident> = <expression>
<expression> ::= <first term> (<addop> <term>)*
<first term> ::= <first factor> <rest>
<term> ::= <factor> <rest>
<rest> ::= (<mulop> <factor>)*
<first factor> ::= [<addop>] <factor>
<factor> ::= <var> | <number> | (<expression>)

The following code implements the BNF:

```
{ Parse and Translate a Math Factor }
procedure Expression ; Forward ;
procedure Factor ;
begin
i f Look = ' ( ' then begin
Match ( ' ( ' ) ;
Expression ;
Match ( ' ) ' ) ;
end
else i f IsAlpha ( Look ) then
LoadVar (GetName)
else
LoadConst (GetNum) ;
end ;
{ Parse and Translate a Negative Factor }
procedure NegFactor ;
begin
Match ( '□' ) ;
i f IsDigit ( Look ) then
LoadConst(□GetNum)
else begin
Factor ;
Negate ;
end ;
end ;
{ Parse and Translate a Leading Factor }
procedure Fi r s tFa c t o r ;
begin
case Look of
'+ ' : begin
Match ( '+ ' ) ;
Factor ;
end ;
'□' : NegFactor ;
else Factor ;
end ;
end ;
{ Recognize and Translate a Mu l t i p l y }
procedure Multiply ;
begin
Match ( ' _ ' ) ;
Factor ;
PopMul ;
end ;
{ Recognize and Translate a Divide }
```

```
procedure Divide ;
begin
Match ( ' / ' ) ;
Factor ;
PopDiv ;
end ;
{ Common Code Used by Term and Fi rstTerm }
procedure Term1 ;
begin
while IsMulop ( Look ) do begin
Push ;
case Look of
' _ ' : Multiply ;
' / ' : Divide ;
end ;
end ;
end ;
{ Parse and Translate a Math Term }
procedure Term;
begin
Factor ;
Term1 ;
end ;
{ Parse and Translate a Leading Term }
procedure Fi rstTerm ;
begin
FirstFactor ;
Term1 ;
end ;
{ Recognize and Translate an Add }
procedure Add ;
begin
Match ( '+ ' ) ;
Term;
PopAdd ;
end ;
{ Recognize and Translate a Subt ract }
procedure Subt ract ;
begin
Match ( '□' ) ;
Term;
PopSub ;
end ;
{ Parse and Translate an Expression }
procedure Expression ;
begin
Fi rstTerm ;
while Is Addop ( Look ) do begin
Push ;
case Look of
'+ ' : Add ;
'□' : Subt ract ;
end ;
end ;
end ;
{ Parse and Translate an Assignment Statement }
procedure Assignment ;
var Name: char ;
begin
Name := GetName ;
Match ( '= ' ) ;
Expression ;
Store (Name) ;
end ;
```

On compiling this we find a reasonable-looking code, representing a complete program that will assemble and execute. We have a compiler!

BOOLEANS

The next step should also be familiar to you. We must add Boolean expressions and relational operations.To begin, we're going to need some more recognizers:

```
{ Recognize a Boolean Orop }
function IsOrop ( c : char ) : boolean ;
begin
IsOrop := c in [ ' | ' , '~ ' ] ;
end ;
{ Recognize a Relop }
function IsRelop ( c : char ) : boolean ;
begin
IsRelop := c in [ '= ' , ' # ' , '< ' , '> ' ] ;
end ;
```

Also, we're going to need some more code generation routines:

```
{ Complement the Primary Register }
procedure No t I t ;
begin
EmitLn ( 'NOT D0 ' ) ;
end ;
```

```
{ AND Top of Stack wi th Primary }
procedure PopAnd ;
begin
EmitLn ( 'AND (SP)+ ,D0 ' ) ;
end ;
{ OR Top of Stack wi th Primary }
procedure PopOr ;
begin
EmitLn ( 'OR (SP)+ ,D0 ' ) ;
end ;
{ XOR Top of Stack wi th Primary }
procedure PopXor ;
begin
EmitLn ( 'EOR (SP)+ ,D0 ' ) ;
end ;
{ Compare Top of Stack wi th Primary }
procedure PopCompare ;
begin
EmitLn ( 'CMP (SP)+ ,D0 ' ) ;
end ;
{ Set D0 I f Compare was = }
procedure SetEqual ;
begin
EmitLn ( 'SEQ D0 ' ) ;
EmitLn ( 'EXT D0 ' ) ;
end ;
{ Set D0 I f Compare was != }
procedure SetNEqual ;
begin
EmitLn ( 'SNE D0 ' ) ;
EmitLn ( 'EXT D0 ' ) ;
end ;
{ Set D0 I f Compare was > }
procedure SetGreater ;
begin
EmitLn ( 'SLT D0 ' ) ;
EmitLn ( 'EXT D0 ' ) ;
end ;
{ Set D0 If Compare was < }
procedure SetLess ;
begin
EmitLn ( 'SGT D0 ' ) ;
EmitLn ( 'EXT D0 ' ) ;
end ;
\ end{ l s t l i s t l i n g }
```

A l l of t h i s gives us the t o o l s we need . The BNF for the Boolean expressions is :

\ begin { verbat im }

<bool□expr > : : = <bool□term> (<orop> <bool□term>)_

<bool□term> : : = <not□fac tor > (<andop> <not□fac tor >)_

<not□fac tor > : : = [' ! '] < r e l a t i o n >

< r e l a t i o n > : : = <expression > [<relop > <expression >]

\ end{ verbat im }

Sharp□eyed readers might note t h a t t h i s syntax does not include the non□terminal " bool□f a c t o r " used in e a r l i e r versions . I t was needed then because I also allowed for the Boolean constants TRUE and FALSE. But remember t h a t in STAT there is no d i s t i n c t i o n made between Boolean and arithme t i c types . . . they can be freely intermi xed . So there is r e a l l y no need for these predef ined values . . . we can j u s t use □1 and 0 , respectively .

In C terminology, we could always use the def ines :

\ begin { verbat im }

def ine TRUE □1

def ine FALSE 0

\ end{ verbat im }

(That is , if STAT had a preprocessor .) Later on, when we al low for declarations of constants, these two values will be predefined by the language .The reason tha t I 'm harping on this is that I' ve already tried the alternative, which is to include TRUE and FALSE as keywords. The problem with that approach is that it then requires lexical scanning for EVERY variable name in every expression. As long as keywords can't be in expressions, we need to do the scanning only at the beginning of every new statement . . . q ui te an improvement . So using the syntax above not only simplifies the parsing , but speeds up the scanning as well, given that we'reall satisfied with the syntax above , the corresponding code i s shown below :

```
\ begin { l s t l i s t i n g } { }
{ Recognize and Translate a Re l a t i o n a l " Equals " }
procedure Equals ;
begin
Match ( '= ' ) ;
Expression ;
PopCompare ;
SetEqual ;
end ;
{ Recognize and Translate a Relational " Not Equals " }
procedure NotEquals ;
begin
Match ( ' # ' ) ;
Expression ;
PopCompare ;
SetNEqual ;
end ;
{ Recognize and Translate a Relational " Less Than " }
procedure Less ;
begin
Match ( '< ' ) ;
Expression ;
PopCompare ;
SetLess ;
end ;
{ Recognize and Translate a Relational " Greater Than " }
procedure Greater ;
141
10.8. BOOLEANS
begin
Match ( '> ' ) ;
Expression ;
PopCompare ;
SetGreater ;
end ;
{ Parse and Translate a Relat ion }
procedure Relat ion ;
begin
Expression ;
i f IsRelop ( Look ) then begin
Push ;
case Look of
'= ' : Equals ;
' # ' : NotEquals ;
'< ' : Less ;
'> ' : Greater ;
end ;
end ;
end ;
{ Parse and Translate a Boolean Factor wi th Leading NOT }
procedure NotFactor ;
begin
i f Look = ' ! ' then begin
Match ( ' ! ' ) ;
Relat ion ;
No t I t ;
end
else
Relat ion ;
end ;
{ Parse and Translate a Boolean Term }
procedure BoolTerm ;
begin
NotFactor ;
whi le Look = '& ' do begin
Push ;
Match ( '& ' ) ;
NotFactor ;
PopAnd ;
end ;
end ;
{ Recognize and Translate a Boolean OR }
procedure BoolOr ;
begin
Match ( ' | ' ) ;
BoolTerm ;
PopOr ;
end ;
{ Recognize and Translate an Exclusive Or }
procedure BoolXor ;
begin
Match ( '~ ' ) ;
BoolTerm ;
PopXor ;
end ;
```

```
{ Parse and Translate a Boolean Expression }
procedure BoolExpression ;
begin
BoolTerm ;
whi le IsOrOp ( Look ) do begin
Push ;
case Look of
' | ' : BoolOr ;
'~ ' : BoolXor ;
end ;
end ;
end ;
```

CONTROL STRUCTURES

We're almost home. With Boolean expressions in place, it's a simple matter to add control structures.

For STAT, we'll only allow two kinds of them, the IF and the WHILE:

<if> ::= IF <bool-expression> <block> [ELSE <block>] ENDIF

<while> ::= WHILE <bool-expression> <block> ENDWHILE

Code for conditional and unconditional branches:

```
{ Branch Unconditional }
procedure Branch ( L : st r ing ) ;
begin
EmitLn ( 'BRA ' + L ) ;
end ;
{ Branch False }
procedure BranchFalse ( L : st r ing ) ;
begin
EmitLn ( 'TST D0 ' ) ;
EmitLn ( 'BEQ ' + L ) ;
end ;
{ Recognize and Translate an IF Const ruct }
procedure Block ; Forward ;
procedure DoI f ;
var L1 , L2 : st r ing ;
begin
Match ( ' i ' ) ;
BoolExpression ;
L1 := NewLabel ;
L2 := L1 ;
BranchFalse ( L1 ) ;
Block ;
i f Look = ' l ' then begin
Match ( ' l ' ) ;
L2 := NewLabel ;
Branch ( L2 ) ;
PostLabel ( L1 ) ;
Block ;
end ;
PostLabel ( L2 ) ;
Match ( ' e ' ) ;
end ;
{ Parse and Translate a WHILE Statement }
procedure DoWhile ;
var L1 , L2 : st r ing ;
begin
Match ( 'w ' ) ;
L1 := NewLabel ;
L2 := NewLabel ;
PostLabel ( L1 ) ;
BoolExpression ;
BranchFalse ( L2 ) ;
Block ;
Match ( ' e ' ) ;
Branch ( L1 ) ;
PostLabel ( L2 ) ;
end ;
```

To tie everything together, we need only modify procedure Block to recognize the "keywords" for the IF and WHILE. As usual, we expand the definition of a block like so:

<block> ::= (<statement>)*

where

<statement> ::= <if> | <while> | <assignment>

The corresponding code is:

```
{ Parse and Translate a Block of Statements }
procedure Block ;
begin
while not ( Look in [ ' e ' , ' l ' ] ) do begin
case Look of
' i ' : DoI f ;
'w ' : DoWhile ;
else Assignment ;
end ;
end ;
end ;
```

OK, add the routines I've given, compile and test them. You should be able to parse the singlecharacter
versions of any of the control constructs. It's looking pretty good!
As a matter of fact, except for the single-character limitation we've got a virtually complete version of STAT.

LEXICAL SCANNING

Now we have to convert the program.To begin with, let's simply allow for whitespace. This involves only adding calls to SkipWhite at the end of the three routines, GetName, GetNum, and Match. A call to SkipWhite in Init primes the pump in case there are leading spaces.Next, we need to deal with newlines. This is really a two-step process, since the treatment of the newlines with single- character tokens is different from that for multi-character ones. We can eliminate some work by doing both steps at once, but I feel safer taking things one step at a time.Insert the new procedure:

```
{ Skip Over an End□of□Line }
procedure NewLine ;
begin
while Look = CR do begin
GetChar ;
i f Look = LF then GetChar ;
SkipWhite ;
end ;
end ;
\ end{ lstlistling }
\ begin { lstistling } { }
{ Type Dec larat ions }
type Symbol = st r ing [ 8 ] ;
SymTab = array [ 1 . . 1 0 0 0 ] of Symbol ;
TabPtr = ^SymTab ;
{ Var iable Dec larat ions }
var Look : char ; { Lookahead Character }
Token : char ; { Encoded Token }
Value : string [ 16 ] ; { Unencoded Token }
ST: Array [ 'A ' . . 'Z ' ] of char ;
{ De f i n i t i o n of Keywords and Token Types }
const NKW = 9;
NKW1 = 10;
const KWl ist : array [ 1 . .NKW] of Symbol =
( ' IF ' , 'ELSE ' , 'ENDIF ' , 'WHILE ' ,
'ENDWHILE ' ,
'VAR ' , 'BEGIN ' , 'END' , 'PROGRAM' ) ;
const KWcode: st r ing [NKW1] = ' xilewevbep '
;
Next , add the three procedures , also from Par t
VI I :
{ Table Lookup }
function Lookup (T : TabPtr ; s : st r ing ; n : i n t
e g e r ) : i n t e g e r ;
var i : i n t e g e r ;
found : Boolean ;
begin
found := f a l s e ;
i := n ;
while ( i > 0) and not found do
i f s = T^ [ i ] then
found := t rue
else
dec ( i ) ;
Lookup := i ;
end ;
..{
Get an Identifier and Scan it for Keywords }
procedure Scan ;
begin
GetName ;
Token := KWcode[ Lookup (Addr ( KWl ist ) ,
Value , NKW) + 1 ] ;
end ;
..{
Match a Specific Input String }
procedure MatchString ( x : string ) ;
begin
i f Value <> x then Expected ( ' ' ' ' + x + ' ' ' ' )
;
end ;
```

MULTI-CHARACTER VARIABLE NAMES

First, add the new typed constant:

```
NEntry: integer = 0;
```

Then change the definition of the symbol table as follows:

```
const MaxEntry = 100;
var ST : array [ 1 . . MaxEntry ] of Symbol ;
```

(Note that ST is NOT declared as a SymTab. That declaration is a phony one to get Lookup to work. A SymTab would take up too much RAM space, and so one is never actually allocated.)

Next, we need to replace InTable:

```
{ Look f o r Symbol i n Table }
function InTable ( n : Symbol ) : Boolean ;
begin
InTable := Lookup (@ST, n , MaxEntry ) <> 0;
end ;
```

~ We also need a new procedure ,

AddEntry ,

t

h

a

t

adds a

new ent r

y

to

the table :

```
{ Add a New Ent ry to Symbol Table }
procedure AddEntry (N: Symbol ; T : char ) ;
begin
i f InTable (N) then Abort ( ' Dupl icate I d e n t i f i e r ' + N) ;
i f NEntry = MaxEntry then Abort ( 'Symbol Table F u l l ' ) ;
Inc ( NEntry ) ;
ST[ NEntry ] := N;
```

151

```
SType [ NEntry ] := T ;
end ;
```

This procedure is c a l l e d by Al l o c :

```
{ Al locate Storage f o r a Var iable }
procedure Al loc (N: Symbol ) ;
begin
i f InTable (N) then Abort ( ' Dupl icate Var iable Name ' + N) ;
AddEntry (N, ' v ' ) ;
...
```

Finally, we must change all the routines that currently treat the variable name as a single character.

These include Load Var and Store (just change the type from char to string), and Factor, Assignment, and Decl (just change Value[1] to Value).

One last thing: change procedure Init to clear the array as shown:

```
{ I n i t i a l i z e }
procedure I n i t ;
var i : i n t e g e r ;
begin
for i := 1 to MaxEntry do begin
ST[ i ] := ' ' ;
SType [ i ] := ' ' ;
end ;
GetChar ;
Scan ;
end ;
```

That should do it. Try it out and verify that you can, indeed, use multi-character variable names.

1.10 More Relops

We still have one remaining single-character restriction: the one on relops. Some of the relops are indeed single characters, but others require two. These are '<=' and '>='. I also prefer the Pascal '<>' for "not equals," instead of '#'.If you'll recall, in Part VII I pointed out that the conventional way to deal with relops is to include them in the list of keywords, and let the lexical scanner find them. But, again, this requires scanning throughout the expression parsing process, whereas so far we've been able to limit the use of the scanner to the beginning of a statement.I mentioned then that we can still get away with this, since the multi-character relops are so few and

so limited in their usage. It's easy to just treat them as special cases and handle them in an ad hoc manner.The changes required affect only the code generation routines and procedures Relation and friends.First, we're going to need two more code generation routines:

```
{ Set D0 I f Compare was <= }
procedure Set Less Or Equal ;
```

```
begin
EmitLn ( 'SGE D0 ' ) ;
EmitLn ( 'EXT D0 ' ) ;
end ;
{ Set D0 I f Compare was >= }
procedure SetGreaterOrEqual ;
begin
EmitLn ( 'SLE D0 ' ) ;
EmitLn ( 'EXT D0 ' ) ;
end ;
```

Then, modify the relation parsing routines as shown below:

```
{ Recognize and Translate a Relational " Less Than or Equal " }
procedure LessOrEqual ;
begin
Match ( '= ' ) ;
Expression ;
PopCompare ;
SetLessOrEqual ;
end ;
{ Recognize and Translate a Re l a t i o n a l " Not Equals " }
procedure NotEqual ;
begin
Match ( '> ' ) ;
Expression ;
PopCompare ;
SetNEqual ;
end ;
{ Recognize and Translate a Re l a t i o n a l " Less Than " }
procedure Less ;
begin
Match ( '< ' ) ;
case Look of
'= ' : LessOrEqual ;
'> ' : NotEqual ;
else begin
Expression ;
PopCompare ;
SetLess ;
end ;
end ;
end ;
{ Recognize and Translate a Re l a t i o n a l " Greater Than " }
procedure Greater ;
begin
Match ( '> ' ) ;
i f Look = '= ' then begin
Match ( '= ' ) ;
Expression ;
PopCompare ;
SetGreaterOrEqual ;
end
else begin
Expression ;
PopCompare ;
SetGreater ;
end ;
end ;
```

That's all it takes. Now we can process all the relops.

CONCLUSION

At this point we have STAT completely defined. It's not much ... actually a toy compiler. STAT has only one data type and no subroutines ... but it's a complete, usable language.

ACKNOWLEDGMENT

I would like to thank my guide Mrs. Reema Sachdeva for her indispensible ideas and continuous support ,encouragement ,advice and understanding me through my difficult times and keeping up my enthusiasm, encouraging me and building up my confidence during the completion of this work. A special thank u note to my parents for their infinite supply of love which kept me going. They always inspired me to follow my instincts.

REFERENCES

1. Andrew W. Appel. Modern Compiler Implementation in C. Cambridge University Press.
2. Apple Computer, Inc.. Dylan Reference Manual.

3. Ken Arnold, James Gosling, and David Holmes. The Java Programming Language. Prentice Hall,
4. Jonathan Bachrach and Keith Playford. The Java Syntactic Extender (JSE). Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 31-42, Tampa Bay, Florida, United States.
5. Jason Baker and Wilson C. Hsieh. Maya: Multiple Dispatch Syntax Extension in Java. Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pages 270-281, Berlin, Germany.
6. Alan Bawden. Quasiquotation in Lisp. Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM '99), pages 4-12, San Antonio, Texas, United States.
7. Thomas Anthony Bergan. Typmix: A Framework For Implementing Modular, Extensible Type Systems. Master's thesis, University of California Los Angeles
8. Computer History Museum. Fellow Awards | 1997 Recipient Dennis Ritchie. Available online at http://www.computer-history.org/fellowawards/index.php?id=71
9. William Clinger and Jonathan Rees. Macros that Work. Proceedings of the 1991 ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages, pages 155-162.
10. Coverity. Linuxbugs. http://linuxbugs. coverity.com Offline. Available via archive.org at http://web.archive.org/web/*/http://linuxbugs.coverity.com
11. Brad J. Cox and Andrew J. Novobilski. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley.
12. Glen Ditchfield. An Overview of Cforall. Available online at http://plg.uwaterloo.ca/~cforall.
13. Eelco Dolstra and Eelco Visser. Building Interpreters with Rewriting Strategies. Proceedings of the 2002 Workshop on Language Descriptions, Tools, and Applications, Grenoble.
14. Gabriel Dos Reis and Bjarne Stroustroup. Specifying C++ concepts. Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 295-308.
15. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. Lisp and Symbolic Computation 5(4), pages 295-326.
16. Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. Proceedings of the 2001 Symposium on Operating Systems Principles, pages 57-72, Banff.
17. Dawson R. Engler. Incorporating Application Semantics and Control into Compilation. Proceedings of the USENIX Conference on Domain-Specific Languages (DSL '97), October 1997.
18. Bob Flandrena. Alef User's Guide. Plan 9 Programmer's Manual: Volume Two.
19. Bryan Ford. Parsing Expression Grammars: a Recognition-Based Syntactic Foundation. Proceedings of the 2004 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy.
20. James Gosling. Ace: a syntax-driven C preprocessor, 1989. Available online at http://swtch.com/gosling89ace.pdf