34	*variable*	→	**id**
35	*variable*	→	**id [** *expression* **]**

34 *variable* → **id**

Notes:

1. The *id* could be a variable
2. The *id* could be a function. For example, consider function g
   function g:real;begin g:=9.8 end;
   The return value of the function is assigned by means of an assignment statement.
3. The type of the expression is the type of the *id*. Even though the expression returned computes the address of the value to be assigned, for the purpose of type-checking, we make the expression type equal to the type of the variable given by parameter *id*.

Exp variable*(*string* id*)

1. Find the symbol descriptor *S* of the *id* in the symbol table.
   Sym* S=ST.Find(*id);
2. Print an error if the *id* cannot be found.
3. Obtain the symbol kind *sk*.
   symkind sk=S->Symkind();
4. switch (sk) {
   case sk_variable: return variable_symbol((VariableSymbol*)S); break;
   case sk_function: return function_symbol((FunctionSymbol*)S); break;
   default: yyerror("Semantic error - ID must be a variable or a function");
   }

Exp* variable_symbol(VariableSymbol* V)

1. Find the difference, *ll*, between the current lexical level and the lexical level of Variable Symbol V.
   int ll=ST.LexicalLevel()-V->LexicalLevel();
2. Create a PCode P
   2.1. Label: none
   2.2. PCode Operation: load address
   lda
   2.3. Operand 1: lexical level difference computed in step 1.
   ll
   2.4. Operand 2: address offset of the variable
   S->Address()
3. Create Expression E
   3.1. Type: Variable Type
   V->Type()
   3.2. PCode: Pcode created in step 2.
   P
4. Return a pointer to the expression created in step 3.
   return E;

Exp* function_symbol(FunctionSymbol* F)

1. Create a PCode P
   1.1. Label: none

   1.2. PCode Operation: load address
   lda
   The address of the return address is at position 0 in the stack mark.
   1.3. Operand 1: 0
   1.4. Operand 2: 0
2. Create Expression E
   2.1. Type: Function Return Type
   F->ReturnType()
   2.2. PCode: Pcode created in step 2.
   P
3. Return a pointer to the expression created in step 3.
   return E;

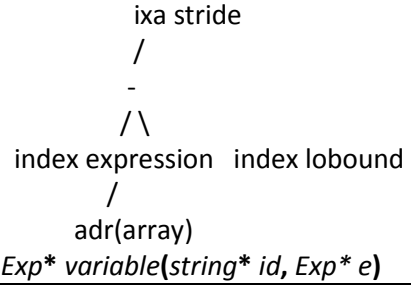
35 *variable* → **id [** *expression* **]**

Notes:

1. The *id* must be the name of an array.
2. The *expression* must have type integer because it is a subscript.
3. The type of the expression returned is given by the element type of the array.

Overview:

1. Find id - id must be an array
2. load the address of id
3. load the value of the expression
4. load the value of the smallest possible index value
5. subtract the index from the expression
6. add the difference to the address of id



*Exp*** variable**(**string*** id**, **Exp* e**)*

1. Find symbol descriptor *S* in the symbol table.
   Sym* S=ST.Find(*id);
2. Issue an error if *id* is not in the symbol table.
3. Issue an error if *id* is not a variablesymbol
4. Issue an error if *id* is not an array.
5. Issue an error if index expression e does not have type integer.
6. Cast variable *V* as a variable symbol of *S*.
   VariableSymbol* V=(VariableSymbol*)S;
7. Load the base address of the array. Create PCode P.
   7.1. Label: none
   7.2. PCode Operation: load address
   lda
   7.3. Operand 1: lexical level difference
   ST.LexicalLevel()-V->LexicalLevel();
   7.4. Operand 2: Address offset
   V->Address();
8. Create an expression R (for right expression)
   8.1. Type: Address
   ST.TAddress()
   8.2. PCode: P created in step 11
   P
9. Append R to the leftmost sub expression of parameter e, the subscript expression.
   e->LeftAppend(R);
10. Compute the unbiased subscript by subtracting the lower bound from the subscript expression.
```
Range* IT=AT->IndexType();
string lo=IT->LoBound();
P=new PCode("","ldc","i",lo);
R=new Exp(ST.TInteger(),P);
P=new PCode("","sbi","","");
L=new Exp(e,R,ST.TInteger(),P);
```
11. Compute the address of element in the array to be assigned.
```
int stride=AT->Stride();
P=new PCode("","ixa","",stride);
Typ* ELT=AT->ElementType();
L=new Exp(L,0,ELT,P);
```
12. Return a pointer to the expression containing the address of the element.
```
return L;
```