The Symbol Table

- The purpose of the symbol table table is to record the use of ‘names’ in a program.
- Such names include:
  - Variables, procedure and function names, constants and user defined types.
- The information stored in the symbol table depends on what the names are used for. For example:
  - A variable name requires its type and runtime address.
  - A procedure requires a pointer the list of arguments it takes.
  - A function requires a pointer to the list of arguments it takes and the return type of the function.
  - An argument requires its type and a pointer to the next argument in the list.

Declarations of Variables

- The purpose of variable declarations in programming languages is to create an entry for that variable in the symbol table and associate a type with it.
- Some programming languages (such as earlier versions of BASIC and APL) do not require a declaration and a symbol table entry is made upon their first use.
- When a compiler meets a statement such as `x = 3`, it must verify that:
  - `x` is declared (look for it in the symbol table),
  - `x` is declared as a variable and not, for example, a procedure name,
  - The type of `x` is an integer or floating-point number.

Functions

- When processing a statement such as

  `if f(a, b) then`, the compiler must check:

  - `f, a` and `b` are declared,
  - `f` takes exactly 2 arguments,
  - `f` is a function and returns a boolean value,
  - `a` and `b` are of the proper types.

Building the Table while Scanning

- When the lexical analyser is scanning the input and meets an identifier, it looks for it in the symbol table:
  - If it **does not find it**, it has to be declared. An entry for that variable is made in the table and its position is returned as the value of the token.
  - If it **does find it**, the position is returned as the value of the token.
- The actual description of the symbol table entry (like its type) is handled by a separate **Object Description Phase**.

Building the Table while Parsing (1)

- A simple lexical analyser does not attempt to process an identifier in anyway. It just returns a token indicating the occurrence of one.
- The actual processing of the identifier is then left to the parser that will deal with it depending on the context in which it has been found. For example, if an identifier is found in a:
  - Declaration Statement, the identifier is looked for in the symbol table. If it is found then the compilers should complain that there is a variable re-declaration. If it does not find it, an entry is made according to the description in the declaration.
  - If the identifier is used in an action statement, a check has to be made to see if it has been declared and that it is used properly (correct number of arguments, no type mismatches, etc…)

Building the Table while Parsing (2)

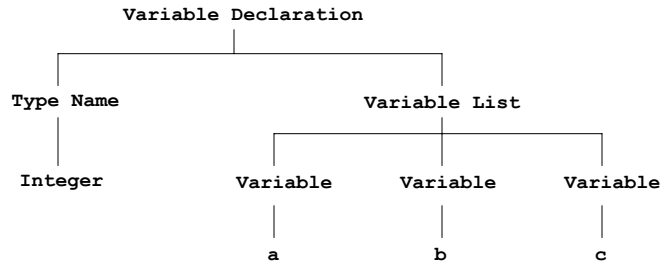
- The method described so far may be implemented in two different ways:
  - The whole parse tree for the variable declaration is built, then declarations in the symbol table and other actions are performed from the tree by the Object Description Phase.
  - Symbol table declarations and other actions are made along the way when parsing.

Symbol Table from the Parse Tree – Option 1

- Suppose we are parsing the declaration:

```
integer a, b, c
```

- The parse tree is passed to an object description phase to analyse it and make the declarations:



Revised Variable Declaration (1) – Option 2

- Note: See slide 67 for original version.
- Consider variable declarations following this format:

```
integer i,j,k
boolean isReady
```

- Recall the grammar:

```
<VarDecl>       ::= <TypeName> <VarNameList>;
<TypeName>      ::= INTEGER|BOOLEAN|REAL;
<VarNameList>   :: <VarName> {“,” <VarName>};
```

Revised Variable Declaration (2)

- Parsing the declaration per se remains the same, so we have no changes so far:

```
Function Parse_VarDecl(TOKEN)
  // Parse the type name
  LOOKAHEAD = Parse_TypeName(TOKEN)

  // Parse the variable name list
  LOOKAHEAD = Parse_VarNameList(LOOKAHEAD)

  Return LOOKAHEAD
End Function
```


Revised Variable Declaration (3)

- Apart from returning the next token, we need to return the type name we just parsed to use later:

```
Function Parse_TypeName(TOKEN)
  If TOKEN is INTEGER_TOKEN then
   Return NextToken, Return Integer Type
  ElseIf TOKEN is BOOLEAN_TOKEN then
   Return NextToken, Return Boolean Type
  ElseIf TOKEN is REAL_TOKEN then
   Return NextToken, Return Real Type
  Else
   Print "Missing Type Name in Declaration"
   Return Error, Return ERROR Type
  End If
End Function
```

Revised Variable Declaration (4)

- Note that when parsing the type name, apart from returning the next lookahead, we also return an indication of which type name we just parsed.
- Parsing the Variable Name List:

```
Function Parse_VarNameList(TOKEN, THE TYPE)
  LOOKAHEAD = Parse_VarName(TOKEN, THE TYPE)
  While LOOKAHEAD = COMMA_TOKEN
   LOOKAHEAD = NextToken
   LOOKAHEAD = Parse_VarName(LOOKAHEAD, THE TYPE)
  End While

  Return LOOKAHEAD
End Function
```


Revised Variable Declaration (4)

- Parsing the variable name:

```
Function Parse_VarName(TOKEN, THE TYPE)
  If TOKEN = IDENT_TOKEN then
   Call procedure VARDECLARE(VARNAME, VARTYPE)
   Note: VARDECLARE is part of the Object
   description phase NOT the parser.
   Return NextToken
  else
   Print "Missing Identifier"
   Return Error
  End If
End Function
```

Final Notes (1)

- The Object Description Phase is a subset of semantic analysis. Ensuring that variables are properly used in action statements is part of another stage of semantic analysis.
- The procedure **VarDeclare**, first looks for the entry of the variable in the symbol table.
  - If the entry is NOT found, a new one is made, recording details (such as the type) of the declaration.
  - If the name is already found, a variable re-declaration might have occurred depending on the **scoping rules** of the language.

Final Notes (2)

- Reuse of the variable declaration is allowed if:
  - All previous uses are no longer in scope.
  - Or, this declaration is made at a lexically lower level than all other active declarations

Final Notes (3)

```
Function VarDeclare(token, VarType)
   Look for a previous occurrence of the Variable

   If no occurrence found then
        Enter details for the variable name and type
   Else
        If      (Use is at a lexically lower level
                than all other active ones) OR
                (Previous uses are not active) then

                Store details in table
        Else
                Re-declaration Error
        End If
   End If
End Function
```