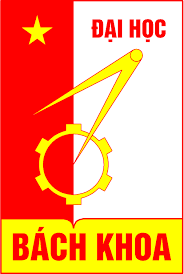
HA NOI UNIVERSITY SCIENCE OF TECHNOLOGY



**COMPILER REPORT**

*Instructor:* **Ts. Nguyễn Thị Thu Hương**

Student: **Lãnh Hùng Sơn**

Student ID: 20092244

Contents

[1. Overview: 2](#_Toc407612327)

[1.1 Purpose of Compiler: 2](#_Toc407612328)

[1.2 Stages in Compiler: 2](#_Toc407612329)

[2. Lexical Analysis ( scanner ): 2](#_Toc407612330)

[2.1 Task of Lexical Analysis ( scanner ): 2](#_Toc407612331)

[2.2 Main Functions in Lexical analysis: 3](#_Toc407612332)

[3. Syntax Analysis ( parser ): 3](#_Toc407612333)

[3.1 Task of Syntax Analysis: 3](#_Toc407612334)

[3.2 Design of Syntax Analysis: 3](#_Toc407612335)

[3.3 Functions of Syntax Analysis: 5](#_Toc407612336)

[4. Semantic Analysis: 6](#_Toc407612337)

[4.1 Task of Semantic Analysis: 6](#_Toc407612338)

[4.2 Design of Symbol table: 6](#_Toc407612339)

[4.3 Components of Symbol table ( symtab ): 7](#_Toc407612340)

[4.3.1 Definition: 7](#_Toc407612341)

[4.3.2 Object: 7](#_Toc407612342)

[4.3.3 Type and Constant: 8](#_Toc407612343)

[4.4 Semantic Rule: 8](#_Toc407612344)

[4.4.1 Check Identifier: 8](#_Toc407612345)

[4.4.2 Check declaration of Identifier: 8](#_Toc407612346)

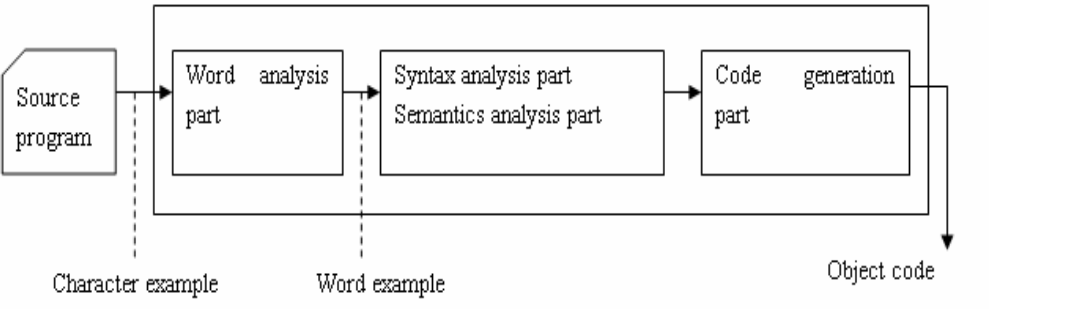
[4.4.3 Check consistency between identifiers defined and identifier using: 8](#_Toc407612347)

# Overview:

## 1.1 Purpose of Compiler:

Compiler is a computer program or a set of computer program to transform code written in high level language to machine language. Result of the program is binary program that called machine code. One of the most important function of compiler is to find error of source code of programmer.

## 1.2 Stages in Compiler:



A compiler program can divide into 4 parts: Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Generation

* Lexical Analysis ( scanner ):
  + Scanning character in source code from left to right and combine them into group of significant word that calls token
* Syntax Analysis ( parser ):
  + Check syntactic structure of a given program
* Semantic Analysis ( semantic ):
  + Check source program to find semantic error and collect information to build code generation stage.
* Code Generation:
  + Generate destination code that includes machine code and assembly

# 2. Lexical Analysis ( scanner ):

## 2.1 Task of Lexical Analysis ( scanner ):

* Skip meaningless characters: blank, tab, new line character, comment.
* Recognize illegal character and return error message.
* Recognize different types of token
  + Identifier
  + Keyword
  + Number
  + Special character
  + ….
* Pass recognized tokens to the parser to perform job of syntactic analysis.

## 2.2 Main Functions in Lexical analysis:

* Void skipBlank(): skip characters: blank character, tab, new line.
* Void skipComment(): skip comment ( string that start by (\* and end by \*) )
* Token \*readIdentKeyword(void): if the first character of string is alphabet, it can be a identifier or keyword.
* Token \*readNumber(void): if the first character of string is number, it is a number
* Token \*readConstChar(void): if the first character of string is singlequote, it is a character
* Token \*getToken(void): read characters from source code and use these functions to create a new token.

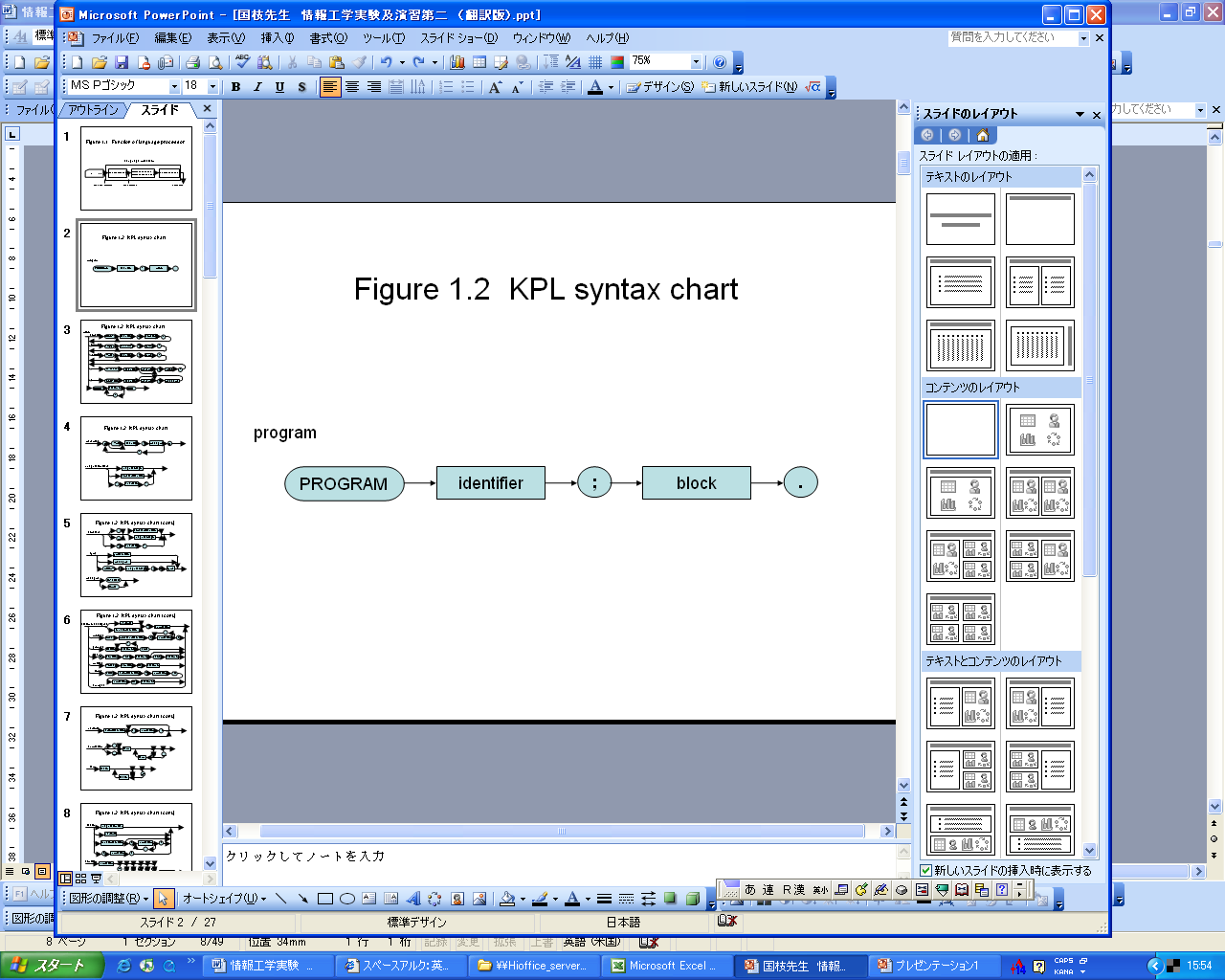
# 3. Syntax Analysis ( parser ):

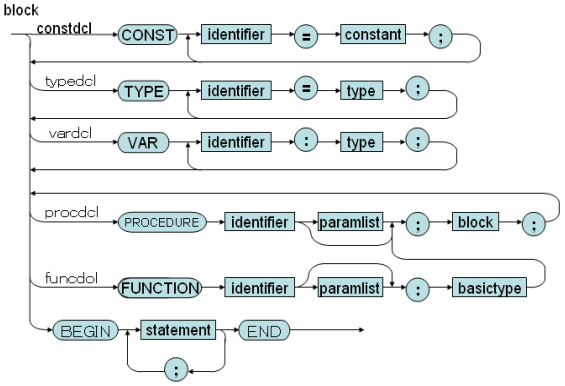
## 3.1 Task of Syntax Analysis:

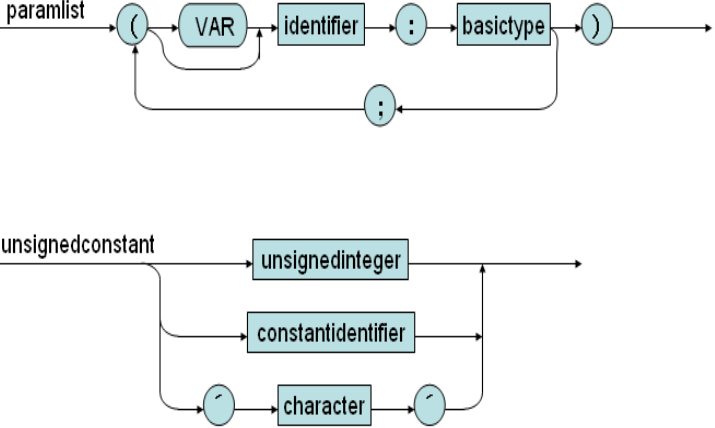
* Check the syntactic structure of a given program
* Invoke semantic analysis and code generation

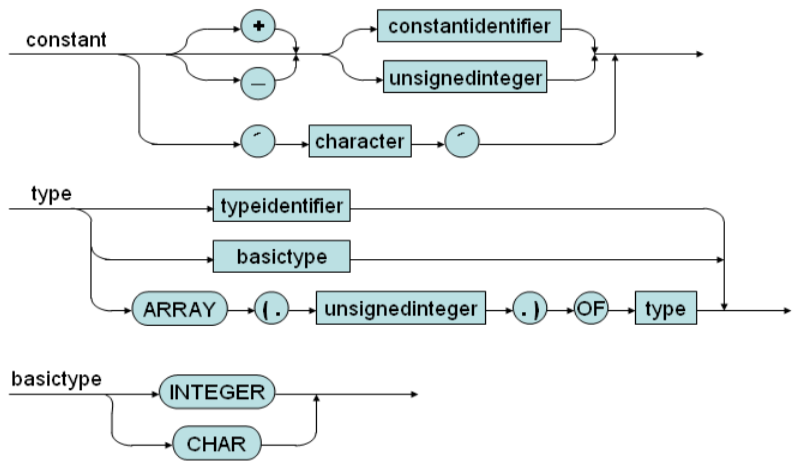
## 3.2 Design of Syntax Analysis:

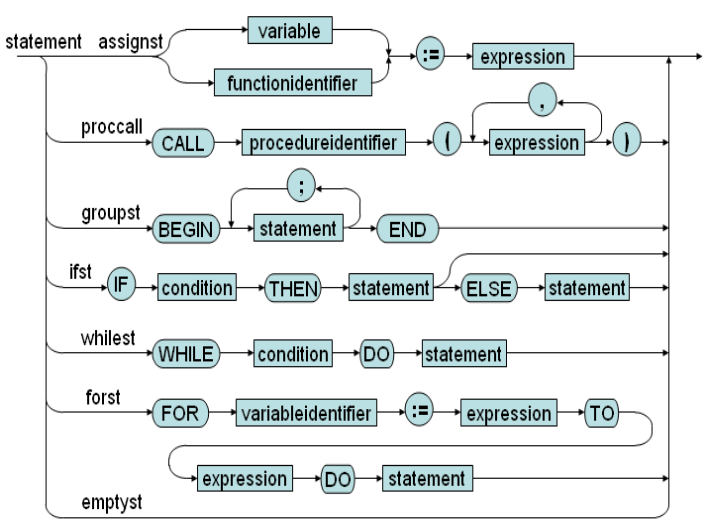
* Use top – down parsing:
  + Token LookAhead
  + Parsing terminal symbol
  + Parsing non-terminal symbol
* Use syntax diagram of KPL:

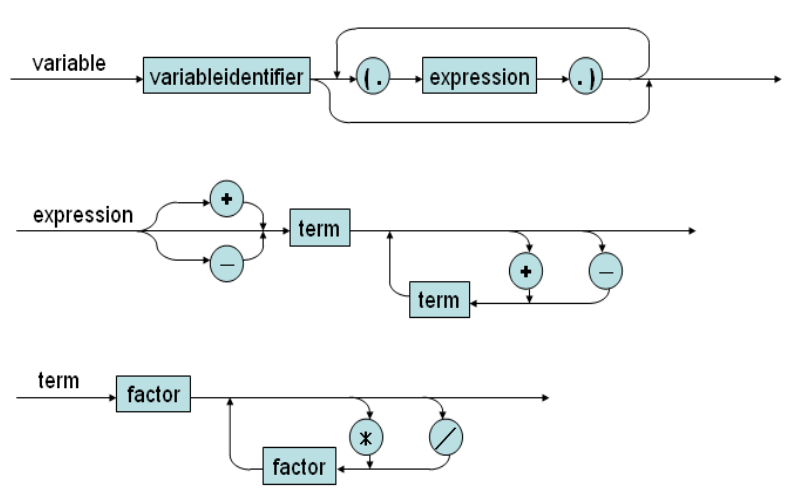


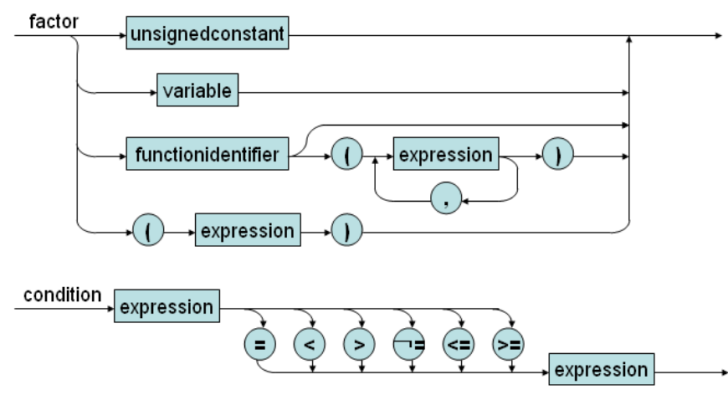


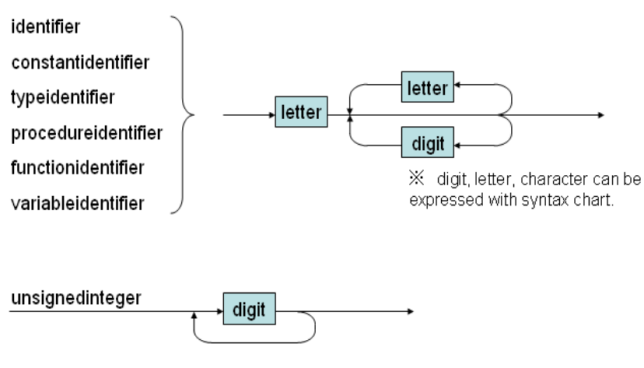












## 3.3 Functions of Syntax Analysis:

* void compileProgram( void ): main program to check
* void compileBlock( void ): parsing block of program
* void compileBlock2( void ): parsing block 2 in block
* void compileBlock3( void ): parsing block 3 in block 2
* void compileBlock4( void ): parsing block 4 in block 3
* void compileBlock5( void ): parsing block 5 in block 4
* void compileConstDecls( void ): parsing declaration of consts
* void compileConstDecls( void ): parsing declaration of a const
* void compileTypeDecls( void ): parsing declaration of types
* void compileTypeDecl( void ): parsing declaration of a type
* void compileVarDecls( void ): parsing declaration of variables
* void compileVarDecl( void ): parsing declaration of a variable
* void compileSubDecls( void ): parsing declaration of sub programs
* void compileFuncDecl( void ): parsing declaration of function
* void compileProcDecl( void ): parsing declaration of procedure
* void compileUnsignedConstant( void ): parsing unsigned constant
* void compileConstant( void ): parsing constant
* void compileType( void ): parsing type
* void compileBasicType( void ): parsing basic type
* void compileParams ( void ): parsing parameters
* void compileParam ( void ): parsing a parameter
* void compileStatements ( void ): parsing statements
* void compileStatement ( void ): parsing a statement
* void compileAssignSt ( void ): parsing assign statement
* void compileCallSt ( void ): parsing call statement
* void compileIfSt ( void ): parsing if statement
* void compileElseSt ( void ): parsing else statement
* void compileWhileSt ( void ): parsing while statement
* void compileForSt ( void ): parsing for statement
* void compileArgument ( void ): parsing argument
* void compileArguments ( void ): parsing arguments
* void compileCondition ( void ): parsing condition
* void compileExpression ( void ): parsing operator ( +, -) and call function compileExpression2
* void compileExpression2 ( void ): parsing operator ( +, -) and call function compileExpression3
* void compileExpression3 ( void ): parsing operator ( +, -) and call function compileTerm
* void compileTerm( void ): parsing term
* void compileFactor( void ): parsing a name, ident, character
* void compileIndexes(): parsing index of array

# 4. Semantic Analysis:

## 4.1 Task of Semantic Analysis:

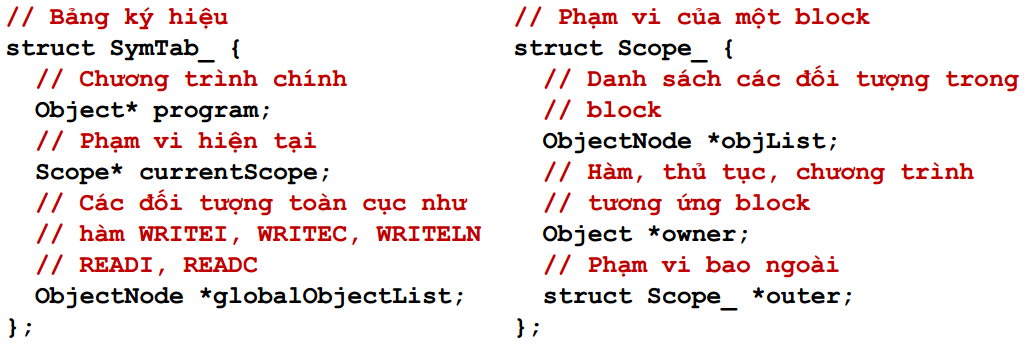
* Manage information about identifier ( const, variable, type that user defined, sub function ( function, procedure ))
* Check law of semantic rule ( range of identifiers, consistency of type )

## 4.2 Design of Symbol table:

* Contain information about identifier and attribute in the program:
  + Const: ( identifier, type, value )
  + Type: is defined by user ( identifier, real type )
  + Variable: identifier, type
  + Function: identifier, parameter, return type, local declaration
  + Procedure: identifier, parameter, local declaration
  + Parameter: identifier, type, value / reference

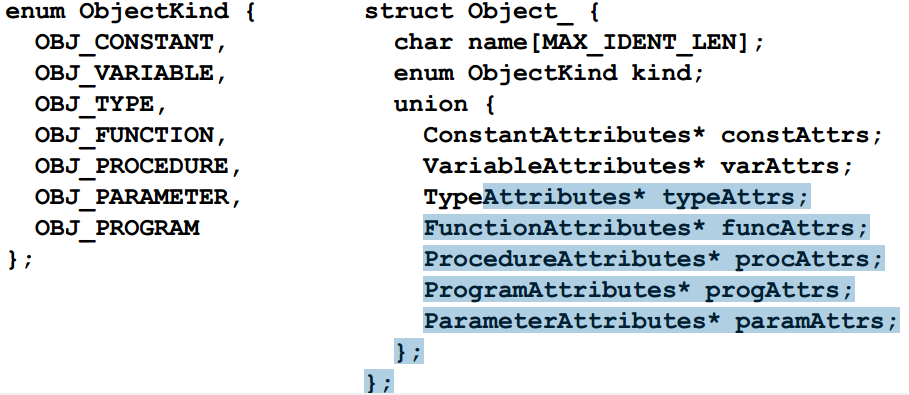
## 4.3 Components of Symbol table ( symtab ):

### 4.3.1 Definition:



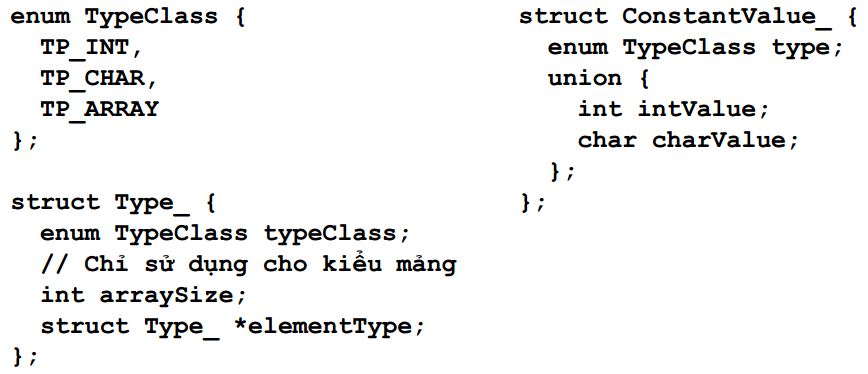
* SymTab contains information about current range of variable in variable currentScope
* Each time check a procedure or a function, we need update this variable by using function void enterBlock( Scope \*scope )
* In the end, we need to change current range by void exitBlock( void )
* To create a new range, we use:
  + Scope \*createScope( Object \*owner, Scope \*outer)

### 4.3.2 Object:



* Object \*createProgramObject( char \*programName ): create new object is new program
* Object \* createConstantObject( char \*name ): create object is constant
* Object \* createTypeObject( char \*name ): create object is type
* Object \*createVariableObject( char \*name ): create object is variable
* Object \*createFunctionObject( char \*name ): create object is function
* Object \*createProcedureObject( char \*name ): create object is procedure
* Object \* create ParameterObject( char \*name, enum ParamKind, Object \*owner ): create object is parameter
* Object \* findObject( ObjectNode \*objList, char \*name ): find a object
* void declareObject( Object \*obj ): declare an object
* void freeObject( Object \*obj ): release an object
* void freeObjectList( ObjectNode \*objList ): release list of object
* void freeReferenceList( ObjectNode \*objList ): release list of object reference
* void addObject( ObjectNode \*\*objList, Object \*obj ): give an object into a list of object

### 4.3.3 Type and Constant:



* Type \*makeIntType( void ): create Integer type
* Type \*makeCharType( void ): create char type
* Type \*makeArrayType( int arraySize, Type \*elementType ): create array type
* Type \*duplicateType( Type \*type ): create a duplicate type
* int compareType( Type \*type1, Type \*type2 ): compare two type
* void freeType( Type \*type ): release a type
* ConstantValue \*makeIntConstant( int I ): create constant interger
* ConstantValue \*makeCharConstant( int I ): create constant char
* ConstantValue \*duplicateConstantValue( ConstantValue \*v ): copy value of constant

## 4.4 Semantic Rule:

### 4.4.1 Check Identifier:

* Void checkFreshIdent( char \*name ): check this identifier existing in current

### 4.4.2 Check declaration of Identifier:

* Check range of identifier:
  + Object \*checkDeclaredIdent( char \*name );
  + Object \*checkDeclaredConstant( char \*name );
  + Object \*checkDeclaredType( char \*name );
  + Object \*checkDeclaredVariable( char \*name );
  + Object \*checkDeclaredType( char \*name );
  + Object \*checkDeclaredFunction( char \*name );
  + Object \*checkDeclaredProcedure( char \*name );
  + Object \*checkDeclaredLValueIdent ( char \*name );

### 4.4.3 Check consistency between identifiers defined and identifier using:

* void checkIntType( Type \*type ): check integer type
* void checkCharType( Type \*type ): check char type
* void checkArrayType( Type \*type ): check array type
* void checkBasicType( Type \*type ): check basic type
* void checkTypeEquality( Type \*type1, Type \*type2 ): compare two type