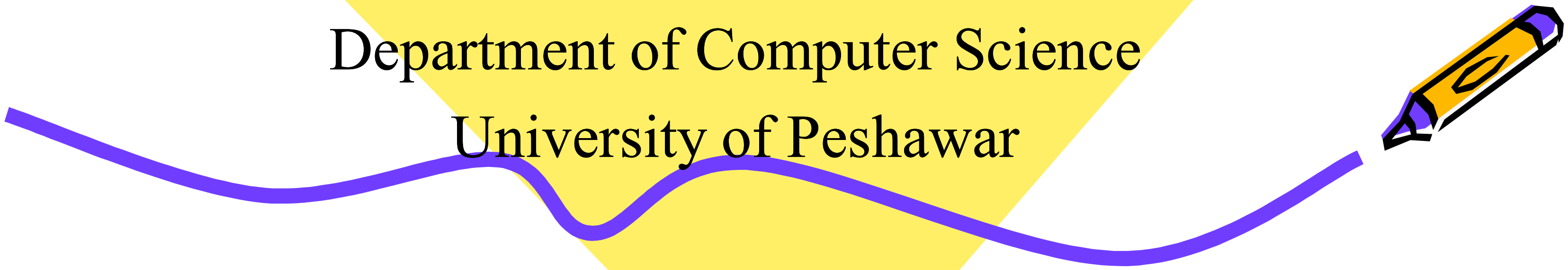
Chapter # 10
Semantic Analyzer

Dr. Shaukat Ali

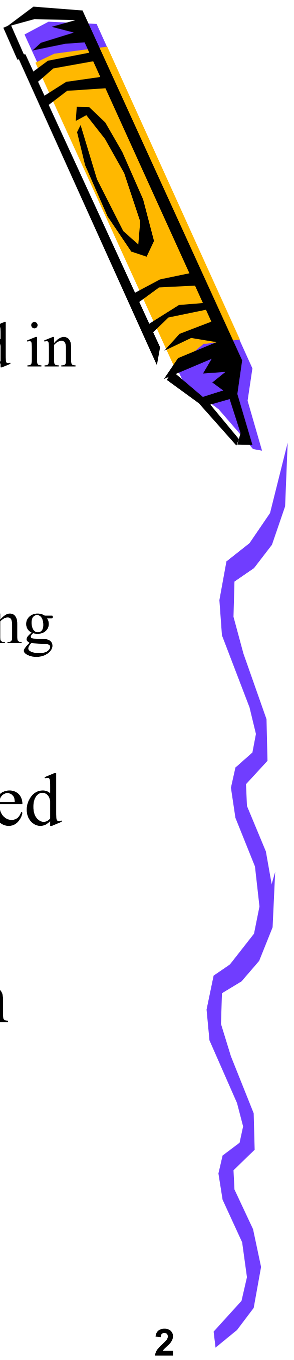
Department of Computer Science

University of Peshawar



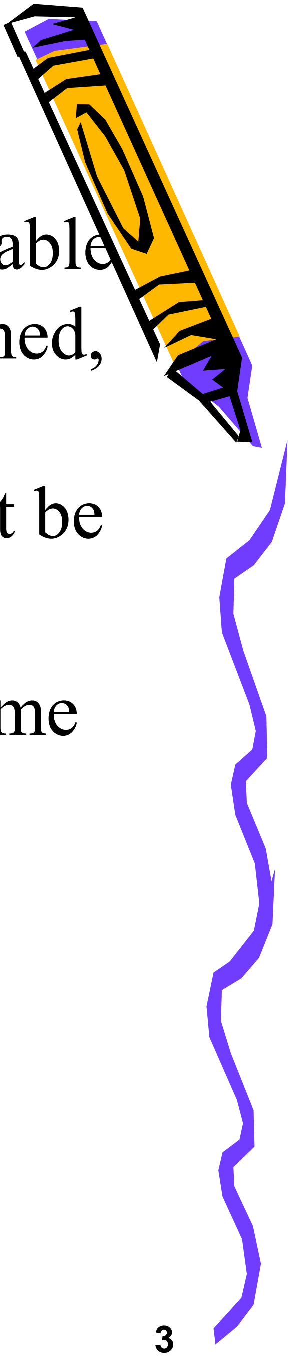
Syntax versus Semantics

- Syntax
  - Verifies that the program consists of tokens arranged in a syntactically valid combination
- Semantics
  - Form a sensible set of instructions in the programming language
- Convention is that syntax is what can be specified by CFG
- Doesn't match intuition - some things that seem to be not definable in CFG
  - Example - number of arguments in function call



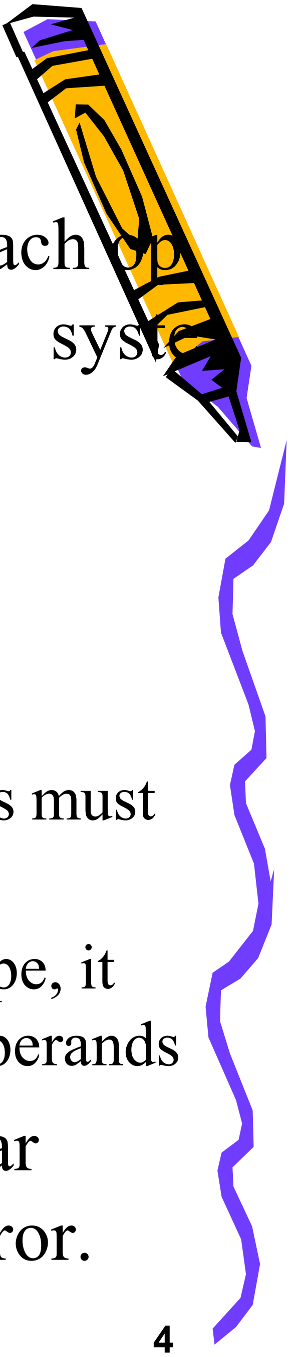
Syntax versus Semantics

- For a program to be semantically valid, all variables, functions, classes, etc. must be properly defined, expressions and variables must be used in ways that respect the type system, access control must be respected, and so forth.
- Static semantics - can be analyzed at compile-time
- Dynamic semantics - analyzed at runtime
  - Division by zero and Array bounds checks
- Not a clear distinction or boundary
- Theory says that while some problems can be found at compile-time, not all can
- So, must have runtime semantic checks

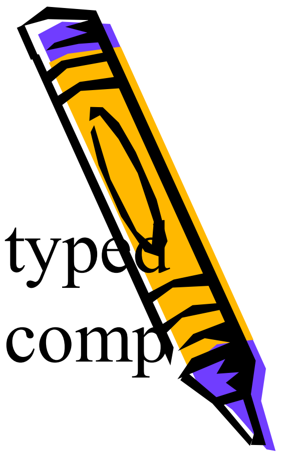


Type Checking

- Type checking is the process of verifying that each operation executed in a program respects the type system of the language.
  - This generally means that all operands in any expression are of appropriate types and number
- For example : x = a % b;
  - % operation requirement is that both of the operands must be of the same type of integer
  - If one is integer and another is of some other data type, it means that % operation is applied to incompatible operands
- If a problem is found, e.g., one tries to add a char pointer to a double in C, we encounter a type error.



Type Checking



- A language is considered strongly typed if each and every type error is detected during comp -ilation.
- Type checking can be done during compilation, during execution, or divided across both.
- Types of type checking
  - Static type checking
  - Dynamic type checking

Static Type Checking

- Static type checking is done at compile time.
- The information the type checker needs is obtained v ia declarations and stored in a master symbol table. A fter this information is collected, the types involved i n each operation are checked.
- It is very difficult for a language that only does static type checking to meet the full definition of strongly typed.
- For example, if a and b are of type int and we assign very large values to the m, a * b may not be in the acceptable range of ints.

Dynamic Type Checking

- Dynamic type checking is implemented by including type information for each data location at run time
  - For example, a variable of type double would contain bot h the actual double value and some kind of tag indicating "double type”
  - The execution of any operation begins by first checking these type tags
  - The operation is performed only if everything checks out. Otherwise, a type error occurs and usually halts exec ution
    - For example, when an add operation is invoked, it first examines the type tags of the two operands to ensure they are compatible.
  - Another example is array out of bound

Dynamic Type Checking

- Normally done in languages that do not require prior data type declaration of variables at compile time
  - For example LISP, JavaScript etc.

- Dynamic type checking clearly comes with a run time performance penalty, but it usually much mo re difficult to subvert and can report errors that are not possible to detect at compile time.

Implicit Type Conversion

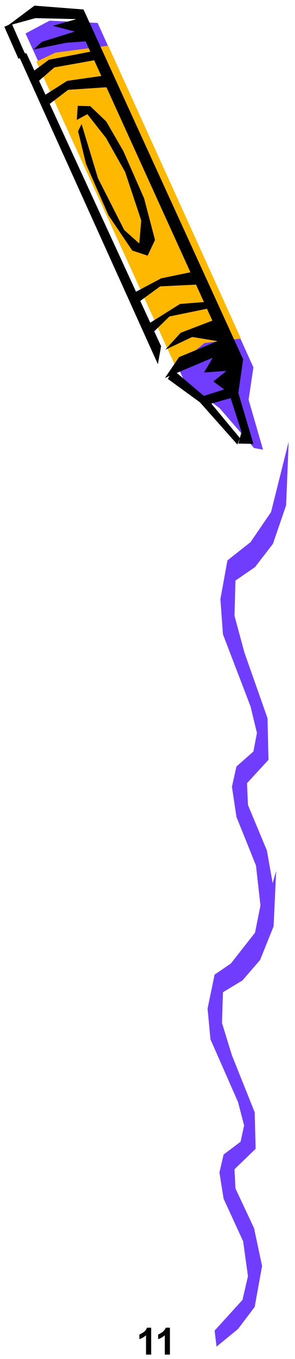
- Many compilers have built-in functionality for cor recting the simplest of type errors.
  Implicit type conversion, or coercion, is when a compiler finds a type error and then changes the type of the variable to an appropriate type.
- This happens in C, for example, when an addition operation is performed on a mix of integer and floating point values.
  - The integer values are implicitly promoted before the a ddition is performed.

Implicit Type Conversion

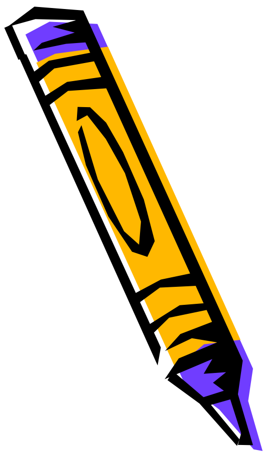
- Other languages are much stricter about type coer
  cion.
  - Ada and Pascal, for example, provide almost no autom
    atic coercions, requiring the programmer to take explic
    it actions to convert between various numeric types.
- The question of whether to provide a coercion ca
  pability or not is controversial.
- Coercions can free a programmer from worrying
  about details, but they can also hide serious errors
   that might otherwise have popped up during com
  pilation.

Types of Static Type Checking

- Four types of static type checks
  - Type checks
  - Flow of control checking
  - Uniqueness Checking
  - Name related checks
- Type Checks
  - To make it sure that the operator is applied to compatible operands
  - A Compiler must report an error if an operator is applied to incompatible operands
  - For example : An integer variable is added with a character variable



Example

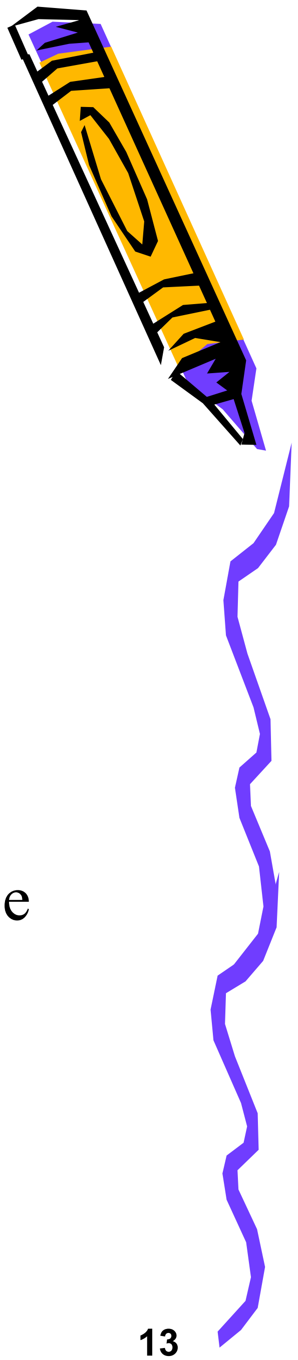


```
int sum;
float a, b;
Sum = a + b;
Product = a * b;
If a.type = int && b.type = int then
Sum.type = int
Else error
If a.type = float && b.type = float then
Sum.type = float
Else error
```

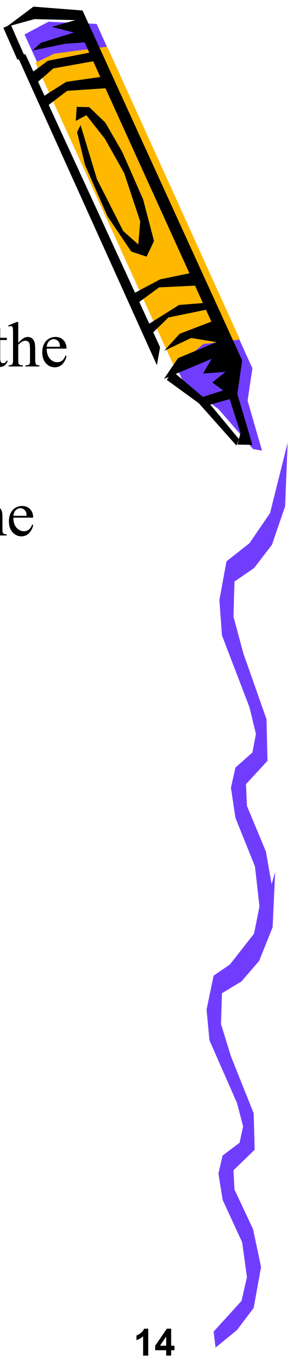
Types of Static Type Checking

- Flow of control checking
  - Checking of branching statements
    - GOTO, BREAK, and CONTINUE etc.
    - If branching location exists or not
  - If it does not exist, then error is reported

- Uniqueness Checking
  - Make it sure unique declaration of variable in a scope
    - No multiple declaration of the same variable in the same scope must exist
      - No two variables with the same name in a scope



Types of Static Type Checking

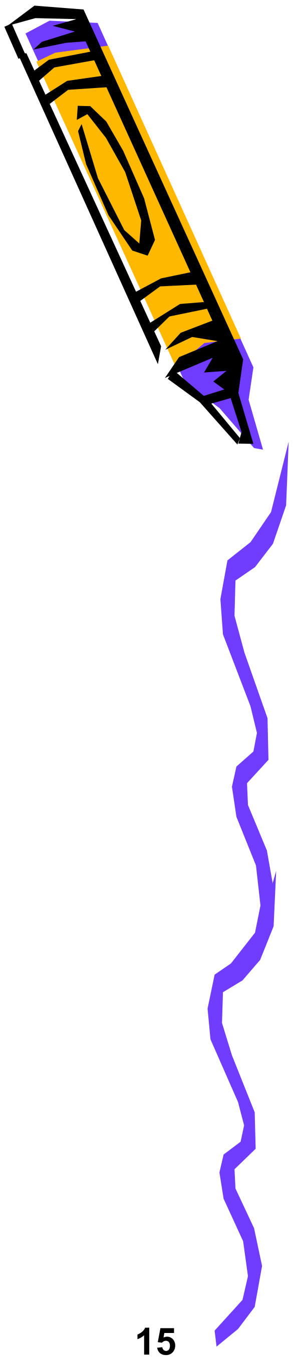


- Name related checks
  - If the same name appears both in the beginning and the ending of a construct
  - It is the duty of compiler to make it sure that the same name appears at both the places
  - Example
    - Unary statement
      - Unit++
      - Unit = unit +1

Coercions

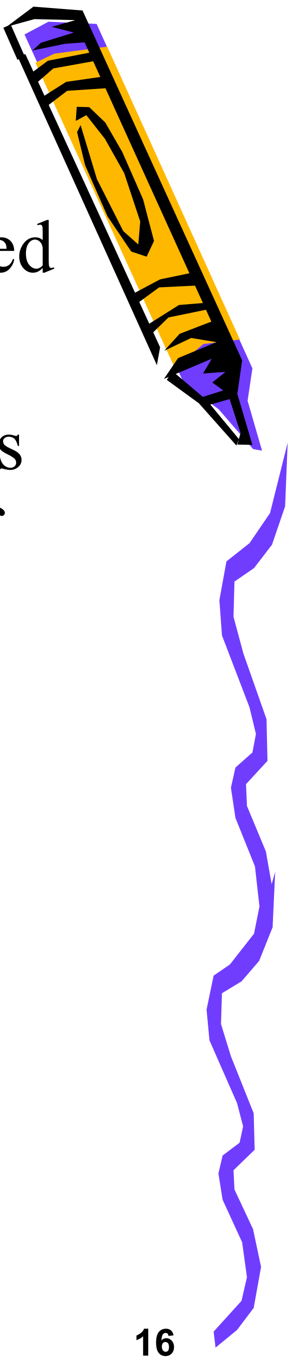
- In case of type mismatch
  - Incompatible operands at the two side s of the an operator
    - The compiler may perform implicit type conversion
    - Called Coercions
- Coercion may occur
  - Incompatibility in assignment statement
  - Incompatible operands of an arithmetic operator

Lvalue		Rvalue
Flaot	=	int



Coercions

- In the first case the type of the RHS is coercioned to the type of the LHS
- In the second case the type having less no of bits is converted into the type having a greater no of bits
- Example in C language char+int
  - Char is converted into int
  - ASCII value of the character is added



Type Checking Rules for Coercions

E → E1 + E2

E.Type = if E1.type == int and E2.type == int then int

Else if E1.type == int and E2.type == real then real

Else if E1.type == real and E2.type == int then real

Else if E1.type == real and E2.type == real then real

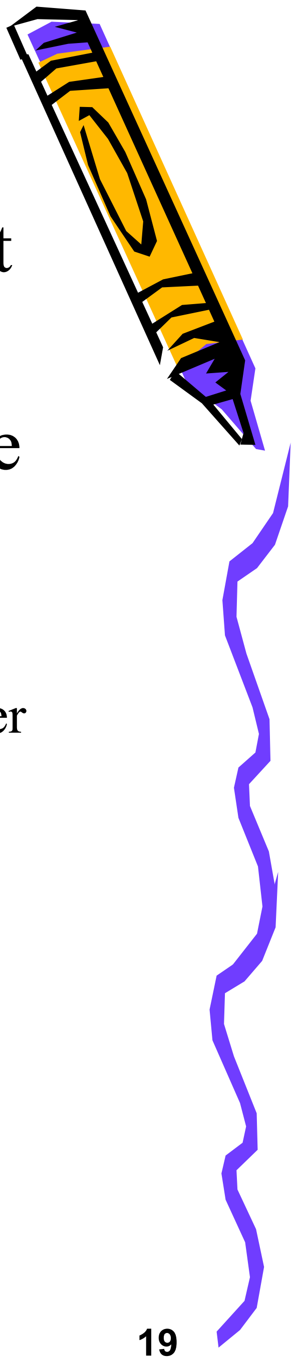
Else type error

Designing a Type Checker

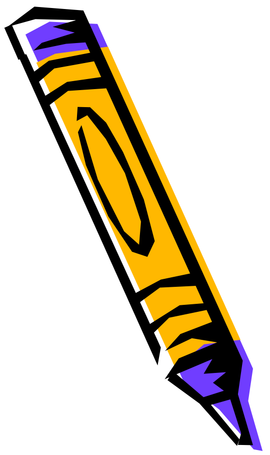
- When designing a type checker for a compiler, here's the process:
  - Identify the types that are available in the language
  - Identify the language constructs that have types associated with them
  - Identify the semantic rules for the language
- we will present it in the context of Decaf
  - Decaf is a somewhat strongly typed language like C
    - Since declarations of all variables are required at compile time.
    - In Decaf, we have base types (**int**, **double**, **bool**, **string**),and compound types (arrays, classes, interfaces).

Designing a Type Checker

- We need to identify the language constructs that have types associated with them.
- In Decaf, here are some of the relevant language constructs:
  - Constants:
    - Obviously, every constant has an associated type. A scanner tells us these types as well as the associated lexeme.
  - Variables:
    - All variables (global, local, and instance) must have a declared type of one of the base types or the supported compound types.
  - Functions:
    - Functions have a return type, and each parameter in the function definition has a type, as does each argument in a function call.



Designing a Type Checker



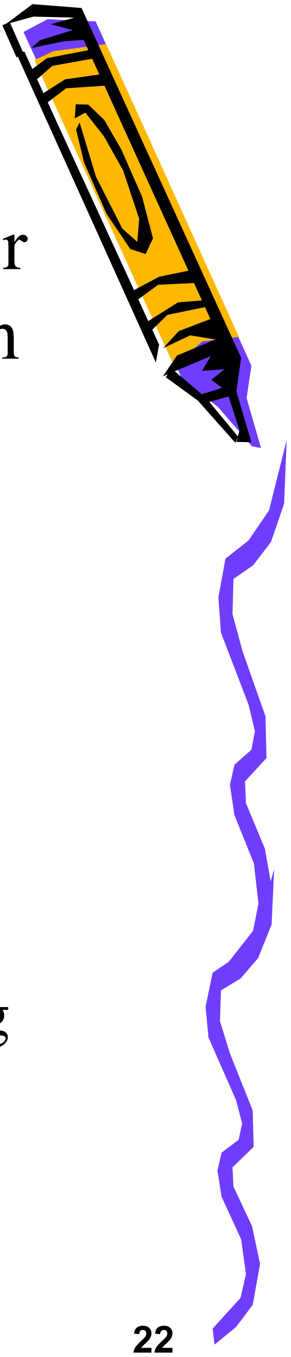
  - Expressions:
    - An expression can be a constant, variable, function call, or some operator (binary or unary) applied to expressions.
    - Each of the various expressions have a type based on the type of the constant, variable, return type of the function, or type of operands.
- Listing the semantic rules that govern what types are allowable in the various language constructs
  - In Decaf, the operand to a unary minus must either be **double** or **int**, the expression used in a loop test must be of **bool** type, and so on.
  - There are also general rules, not just a specific construct, such as all variables must be declared, all classes are global, and so on.

Designing a Type Checker

- These three things together (the types, the relevant constructs, and the rules) define a *type system* for a language.
- Once we have a type system, we can implement a type checker as part of the semantic analysis phase in a compiler.

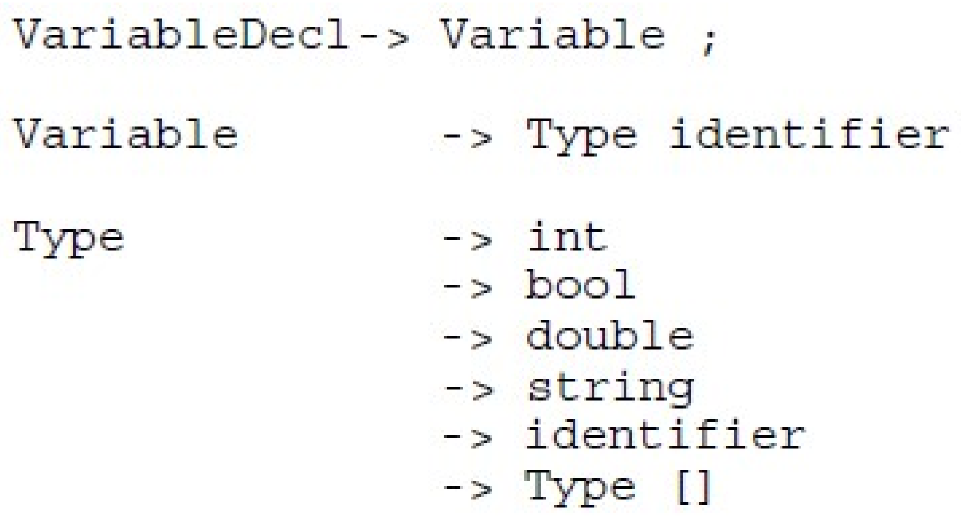
Implementation

- The first step in implementing a type checker for a compiler is to record type information for each identifier
  - All a scanner knows is the name of the identifier so that it what is passed to the parser
  - Typically, the parser will create some sort of "declaration" record for each identifier after parsing. Its declaration record to be stored for later.
  - On encountering uses of that identifier, the semantic analyzer can lookup that name and find the matching declaration or report error when no declaration has been found.



Implementation

– Let’s consider an example. In Decaf, we have the following production rules that are used to parse a variable declaration



```
VariableDecl-> Variable ;

Variable       -> Type identifier

Type           -> int
               -> bool
               -> double
               -> string
               -> identifier
               -> Type []
```

– Consider the following variable declaration

int a;

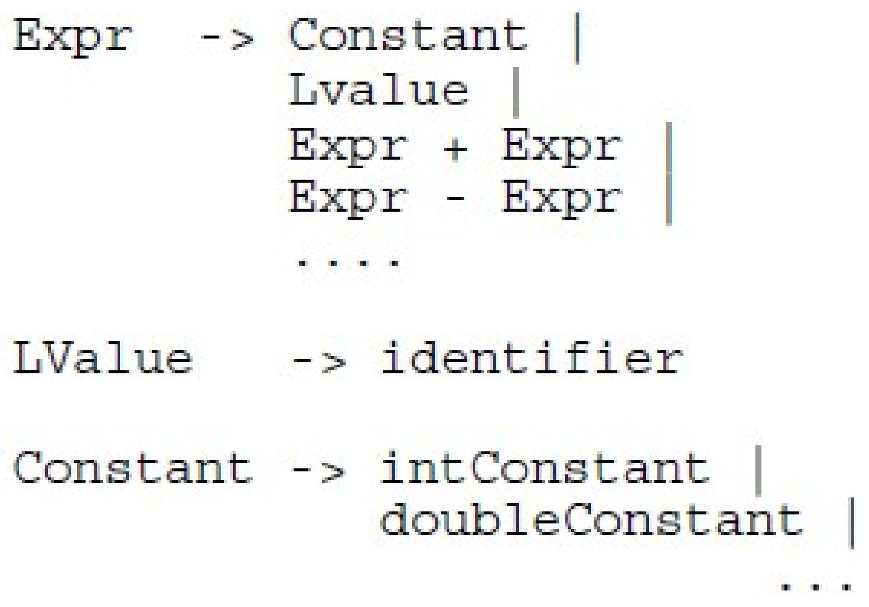
double b;

Implementation

- The scanner records the name for an identifier
- The parser uses the grammar to parse variable declaration
  - The type associated with the **Type** symbol (passed up from the **Type** production)
  - Name associated with the **identifier** symbol (passed from the scanner)
  - Therefore, the parser records new variable declaration, declaring that identifier to be of that type, in a symbol table for lookup later on

- Once we have the type information stored away and easily accessible, we use it to check that the program follows the general semantic rules and the specific ones concerning the language constructs

Implementation



```
Expr     -> Constant |
            Lvalue |
            Expr + Expr |
            Expr - Expr |
            ....

LValue      -> identifier

Constant -> intConstant |
               doubleConstant |
                             ...
```

- In parsing an expression such as **x + 7**
  - Apply the **LValue** and **Constant** productions to the two sides respectively
  - We need the identifier information for the variable on the left and the constant from the right
  -

Implementation

- When we are handling the **Expr** + **Expr** production, we examine the type of each operand to determine if it is appropriate in this context
  - Which in Decaf, means the two operands must be both **int** or both **double**
- The type information will be stored in the symbol table
  - The **X** would of type **int** stored in declaration record in the symbol table by the parser.
  - The **7** would be of type **int** specified in the production rules and stored in the symbol table by the parser.
- The Decaf semantic analyzer typer checker will lookup the symbol table for the type values
  - Which is both **int** in this case and satisfies the semantic rules of the language.

- End of Chapter # 10