CLASSTIME | Page No.
Date / /

# 6.3 Types and Declarations.

The applications of types can be grouped into two :-

## 1.) Type checking :

→ uses logical rules to reason about the behaviour of a program at runtime.

→ Ensures that the types of the operands match the type expected by an operator.

example :
the && operator expects it two operands to be booleans ; the result is also of type boolean.

## 2.) Translation Applications :

→ From the type of a name, a compiler can determine the storage for that name at run time.

→ Type information is needed to :-
- calculate the address denoted by array reference.
- insert explicit type conversions.
- chose right version of operator.

CLASSTIME | Page No
Date / /

**Relative address :** Relative address of a name or a component of a data structure is an offset from the start of a data area.

## 6.3.1 Type Expressions.

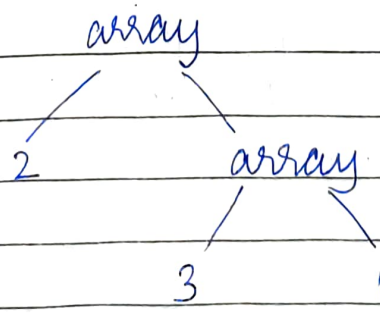
→ Type expressions are used to represent the 'structure of types'.

→ It is either a basic type or is formed by applying an operator called a type constructor to a type expression.

example :

The array type int [2][3] can be read as "array of 2 arrays of 3 integers each" and written as a type expression array (2, array (3, integer)).

can be represented as -




→ operator array takes two parameters - a number and a type.

CLASS/TIME | PAGE NO.
Date / /

## Definition of type expressions :

1. A basic type is a type expression.
   → include boolean, char, integer, float and void.

2. A type name is a type expression.

3. A type expression can be formed by applying the array type constructor to a number and a type expression.

4. A type expression can be formed by applying the record type constructor to the field names and their types.
   (record is a data structure with named fields.)

5. A type expression can be formed by using the type constructor → for function types. We write $s \rightarrow t$ for "function from type s to type t".

6. If s and t are type expressions, then their cartesian product $s \times t$ is also a type expression.

7. Type expressions may contain variables whose values are type expressions.

CLASSTIME | Page No.
Date / /

## 6.3.2 Type Equivalence.

- When type expressions are represented by graphs, two types are structurally equivalent iff one of the following conditions is true :

→ They are the same basic type.

→ They are formed by applying the same constructor to structurally equivalent types

→ One is a type name that denotes the other.

- If type names are treated as standing for themselves, then the first two conditions in the above definition lead to name equivalence of type expressions.

## 6.3.3 Declarations.

Grammar -

D → T id ; D | ∈

T → B C | record '{' D '}'

B → int | float

C → ∈ | [ num ] C

→ Nonterminal D generates a sequence of declarations.

→ Nonterminal T generates basic, array or record types.

→ Nonterminal B generates one of the basic types int and float.

→ Nonterminal C, for "component", generates strings of zero or more integers, each integer surrounded by brackets.

→ An array type consists of a basic type specified by B, followed by array component specified by nonterminal C.

→ A record type (the 2nd production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

## 6.3.4 Storage Layout for Local Names.

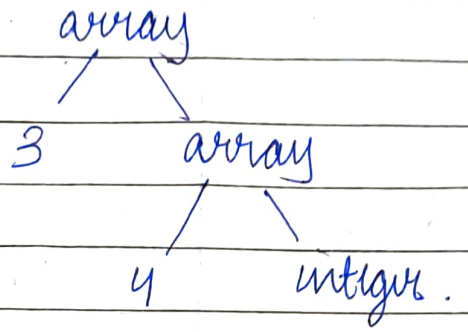
→ Earlier, we considered an SDD for arrays that was able to compute the type. The key point was that it called the function array(size, type) so as to produce a tree structure exhibiting the

dimensionality of the array.

example-

int [3][4]

⇒ array [3] of array [4] of int.




→ Now, we will extend the SDD to calculate the size of the array as well.

For above example, size of the array will be 48, int has size 4.

When we declare many objects, we should know the size of each one in order to determine the offsets of the objects from the start of the storage area.

→ Here, we'll considering only those types for which the storage requirements can be computed at compile time.

For others, eg. string variables, dynamic arrays, etc. we would only be reserving space for a pointer to the structure.

→ Multibyte objects are stored in consecutive bytes and given the address of the first byte.

→ The width of a type is the no. of storage units needed for objects of that type.

| | |
|---|---|
| T → B C | { t = B.type ; w = B.width; } |
| B → int | { B.type = integer; B.width = 4; } |
| B → float | { B.type = float ; B.width = 8 ; } |
| C → ε | { C.type = t; width = w; } |
| C → [num] $C_1$ | { array(num.value , G.type); C.width = num.value × $C_1$. width; } |

Computing types and their width.

↳ The translation scheme (SDT) computes the types and their widths for basic and array types.

↳ The SDT uses synthesized attributes type and width for each non-terminal and two variables t and w to pass type and width information from B node in a parse tree

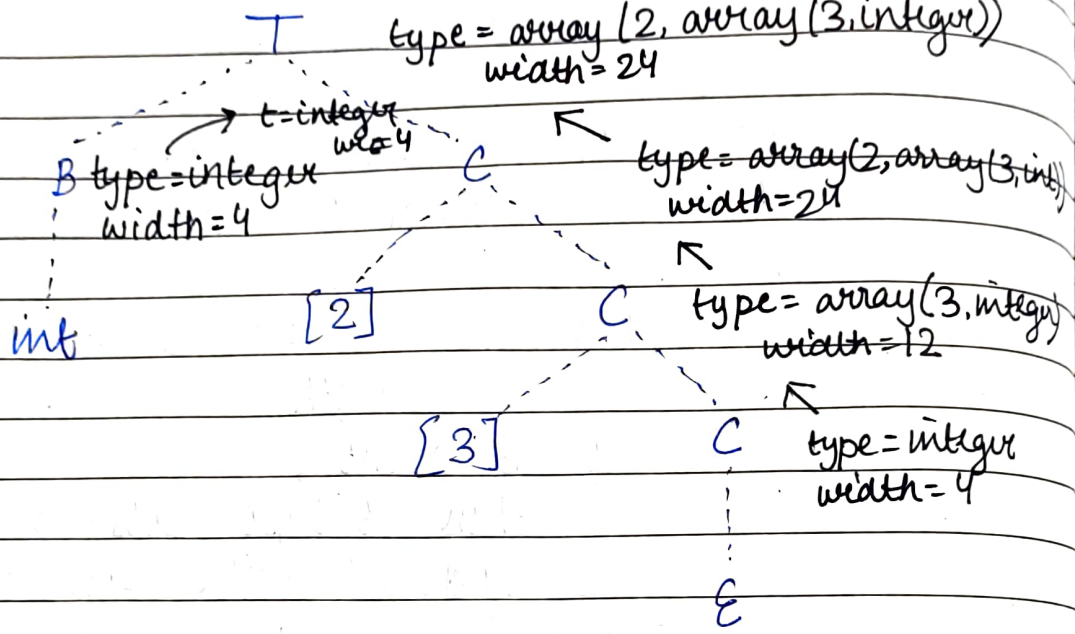
CLASSTIME | Page No.
Date / /

to the node for the production $C \rightarrow \varepsilon$.
- t and w are inherited attributes for C.

- The body of the T-productions consists of non-terminal B, an action and NT C.
  - the action sets t to B.type and w to B.width.
  - if $B \rightarrow$ int,
    then B.type sets to integer
    and, B.width sets to 4.
  - similarly, if $B \rightarrow$ float,
    then B.type sets to float
    and, B.width sets to 8.

- The production for C determines whether T generates a basic type or an array.
  - If $C \rightarrow \varepsilon$
    then, t becomes C.type
    and w becomes C.width.
  - If $C \rightarrow$ [num] $C_1$.
    - specifies an array component.
    - C.type will be formed by applying the type constructor array to the operands.
    - the width is calculated by multiplying the width of an element by the no. of elements in the array.

example:




Syntax-directed translation of array types.

→ parse tree of int [2][3] is shown by dotted lines.

→ solid lines show how the type & width are passed from B, down the chain of C's through variables t and w, and then back up the chain as synthesised attributes type and width.