Data Types

Data Types and Type Information

- Program data can be classified according to their types.
- At the most basic level, virtually every data value expressible in a programming language has an implicit type.
- For example, in C the value 21 is of type int, 3.14159 is of type double, and "hello" is of type “array of char”
- the types of variables are often explicitly associated with variables by a declaration, such as:

```
int x;
```

- which assigns data type int to the variable x.
- In a declaration like this, a type is just a name (in this case the keyword int), which carries with it some properties, such as the kinds of values that can be stored, and the way these values are represented internally.

- Some declarations implicitly associate a type with a name, such as the Pascal declaration

  const PI = 3.14159;

- which implicitly gives the constant PI the data type real, or the ML declaration

  val x = 2;

- which implicitly gives the constant x the data type int.
- Since the internal representation is a system-dependent feature, from the abstract point of view, we can consider a type name to represent the possible values that a variable of that type can hold.

- **DEFINITION : 1.** A data type is a set of values.
- Thus, the declaration of x as an int says that the value of x must be in the set of integers as defined by the language (and the implementation), or, in mathematical terminology (using $\in$ for membership) the declaration

int x;

means the same as:

value of x $\in$ Integers

where Integers is the set of integers as defined by the language and implementation.

- A data type as a set can be specified in many ways: It can be explicitly listed or enumerated, it can be given as a subrange of otherwise known values, or it can be borrowed from mathematics, in which case the finiteness of the set in an actual implementation may be left vague or ignored.
- Set operations can also be applied to get new sets out of old
- A set of values generally also has a group of operations that can be applied to the values.
- These operations are often not mentioned explicitly with the type, but are part of its definition.

- Examples include the arithmetic operations on integers or reals, the subscript operation [ ] on arrays.
- These operations also have specific properties that may or may not be explicitly stated [e.g., (x + 1) − 1 = x or x + y = y + x].
- Thus, to be even more precise, we revise our first definition to include explicitly the operations, as in the following definition:

D E F I N I T I O N : 2. A data type is a set of values, together with a set of operations on those values having certain properties.

- A language translator can make use of the set of algebraic properties in a number of ways to assure that data and operations are used correctly in a program.
- For example, given a statement such as

  z = x / y;

- a translator can determine whether x and y have the same (or related) types, and if that type has a division operator defined for its values (thus also specifying which division operator is meant, resolving any overloading).

- For example, in C and Java, if x and y have type int, then integer division is meant (with the remainder thrown away) and the result of the division also has type int, and if x and y have type double, then floating-point division is inferred.
- Similarly, a translator can determine if the data type of z is appropriate to have the result of the division copied into it.
- In C, the type of z can be any numeric type (and any truncation is automatically applied), while in Java, z can only be a numeric type that can hold the entire result (without loss of precision).

- The process a translator goes through to determine whether the type information in a program is consistent is called type checking.
- In the preceding example, type checking verifies that variables x, y, and z are used correctly in the given statement.
- Type checking also uses rules for inferring types of language constructs from the available type information.
- In the example above, a type must be attached to the expression x / y so that its type may be compared to that of z (in fact, x / y may have type int or double, depending on the types of x and y).
- The process of attaching types to such expressions is called **type inference.**
- Type inference may be viewed as a separate operation performed during type checking, or it may be considered to be a part of type checking itself.

- Given a group of basic types like int, double, and char, every language offers a number of ways to construct more complex types out of the basic types; these mechanisms are called **type constructors**, and the types created are called user-defined types.
- For example, one of the most common type constructors is the array, and the definition

```
int a[10];
```

- creates in C (or C++) a variable whose type is “array of int” (there is no actual array keyword in C), and whose size is specified to be 10.

- New types created with type constructors do not automatically get names.
- Names are, however, extremely important, not only to document the use of new types but also for type checking.
- Names for new types are created using a type declaration (called a type definition in some languages).
- For example, the variable a, created above as an array of 10 integers, has a type that has no name (an anonymous type).
- To give this type a name, we use a typedef in C:

```
typedef int Array_of_ten_integers[10];
Array_of_ten_integers a;
```

- During type checking a translator must often compare two types to determine if they are the same, even though they may be user-defined types coming from different declarations (possibly anonymous, or with different names).
- Each language with type declarations has rules for doing this, called **type equivalence** algorithms.
- The methods used for constructing types, the type equivalence algorithm, and the type inference and type correctness rules are collectively referred to as a **type system**.

- If a programming language definition specifies a complete type system that can be applied statically and that guarantees that all (unintentional) data-corrupting errors in a program will be detected at the earliest possible point, then the language is said to be strongly typed.
- Strong typing ensures that most unsafe programs (i.e., programs with data-corrupting errors) will be rejected at translation time, and those unsafe programs that are not rejected at translation time will cause an execution error prior to any data-corrupting actions.
- Thus, no unsafe program can cause data errors in a strongly typed language.

- C has even more loopholes and so is sometimes called a weakly typed language. C++ has attempted to close some of the most serious type of loopholes of C, but for compatibility reasons it still is not completely strongly typed.

Simple Types

- Algol-like languages (C, Ada, Pascal), including the object-oriented ones (C++, Java), all classify data types according to a relatively standard basic scheme, with minor variations.
- Every language comes with a set of predefined types from which all other types are constructed.
- These are generally specified using either keywords (such as int or double in C++ or Java) or predefined identifiers (such as String or Process in Java).
- Sometimes variations on basic types are also predefined, such as short, long, long double, unsigned int, and so forth that typically give special properties to numeric types.

- Predefined types are primarily simple types: types that have no other structure than their inherent arithmetic or sequential structure.
- However, there are simple types that are not predefined: enumerated types and subrange types are also simple types.
- Enumerated types are sets whose elements are named and listed explicitly. A typical example (in C) is

```
enum Color {Red, Green, Blue};
```

or (the equivalent in Ada):

```
type Color_Type is (Red, Green, Blue);
```

- In C an enum declaration defines a type "enum . . . ," but the values are all taken to be names of integers and are automatically assigned the values 0, 1, and so forth, unless the user initializes the enum values to other integers;
- Thus, the C enum mechanism is really just a shorthand for defining a series of related integer constants, except that a compiler could use less memory for an enum variable.

```
(1) #include <stdio.h>
(2) enum Color {Red,Green,Blue};
(3) enum NewColor {NewRed = 3, NewGreen = 2, NewBlue = 2};
(4) main(){
(5) enum Color x = Green; /* x is actually 1 */
(6) enum NewColor y = NewBlue; /* y is actually 2 */
(7) x++; /* x is now 2, or Blue */
(8) y--; /* y is now 1 -- not even in the enum */
(9) printf("%d\n",x); /* prints 2 */
(10) printf("%d\n",y); /* prints 1 */
(11) return 0;
(12) }
```

Type Constructors

- Since data types are sets, set operations can be used to construct new types out of existing ones.
- Such operations include Cartesian product, union, powerset, function set, and subset.
- When applied to types, these set operations are called type constructors.
- In a programming language, all types are constructed out of the predefined types using type constructors.

Cartesian Product

- Given two sets U and V, we can form the Cartesian or cross product consisting of all ordered pairs of elements from U and V:
- $U \times V = \{(u, v) \mid u \text{ is in } U \text{ and } v \text{ is in } V\}$
- Cartesian products come with projection functions $p1: U \times V \rightarrow U$ and $p2: U \times V \rightarrow V$, where $p1((u, v)) = u$ and $p2((u, v)) = v$.
- This construction extends to more than two sets. Thus $U \times V \times W = \{(u, v, w) \mid u \text{ in } U, v \text{ in } V, w \text{ in } W\}$.
- There are as many projection functions as there are components.

- In many languages the Cartesian product type constructor is available as the record or structure construction. For example, in C the declaration

```
struct IntCharReal{
int i;
char c;
double r;
};
```

constructs the Cartesian product type int × char × double.

- The projections in a record structure are given by the component selector operation (or structure member operation): If x is a variable of type IntCharReal, then x.i is the projection of x to the integers.

Union

- A second construction, the union of two types, is formed by taking the set theoretic union of their sets of values.
- Union types come in two varieties: discriminated unions and undiscriminated unions.
- A union is discriminated if a tag or discriminator is added to the union to distinguish which type the element is—that is, which set it came from.
- Discriminated unions are similar to disjoint unions in mathematics.
- Undiscriminated unions lack the tag, and assumptions must be made about the type of any particular value

- In C (and C++) the union type constructor constructs undiscriminated unions:

```
union IntOrReal{
int i;
double r;
};
```

- Note that, as with structs, there are names for the different components (i and r).
- These are necessary because their use tells the translator which of the types the raw bits stored in the union should be interpreted as; thus, if x is a variable of type union IntOrReal, then x.i is always interpreted as an int, and x.r is always interpreted as a double.
- These component names should not be confused with a discriminant, which is a separate component that indicates which data type the value really is, as opposed to which type we may think it is.

A discriminant can be imitated in C as follows:

```
enum Disc {IsInt,IsReal};
struct IntOrReal{
enum Disc which;
union{
int i;
double r;
} val;
};
```

and could be used as follows:

```
IntOrReal x;
x.which = IsReal;
x.val.r = 2.3;
. . .
if (x.which == IsInt) printf("%d\n",x.val.i);
else printf("%g\n",x.val.r);
```

Of course, this is still unsafe, since it is up to the programmer to generate the appropriate test code (and the components can also be assigned individually and arbitrarily).

Subset

- In mathematics, a subset can be specified by giving a rule to distinguish its elements, such as pos_int = {x | x is an integer and x > 0}.
- Similar rules can be given in programming languages to establish new types that are subsets of known types.
- Ada, for example, has a very general subtype mechanism.
- By specifying a lower and upper bound, subranges of ordinal types can be declared in Ada, as for example:
- subtype IntDigit_Type is integer range 0..9;
- Compare this with the simple type:
- type Digit_Type is range 0..9;
- which is a completely new type, not a subtype of integer.

Arrays and Functions

- The set of all functions f : U → V can give rise to a new type in two ways: as an array type or as a function type.
- When U is an ordinal type, the function f can be thought of as an array with index type U and component type V: if i is in U, then f(i) is the i^{th} component of the array, and the whole function can be represented by the sequence or tuple of its values (f(low), . . . , f(high)), where low is the smallest element in U and high is the largest. (For this reason, array types are sometimes referred to as sequence types.)
- In C, C++, and Java the index set is always a positive integer range beginning at zero.

- Typically, array types can be defined either with or without sizes, but to define a variable of an array type it's typically necessary to specify its size at translation time, since arrays are normally allocated statically or on the stack (and thus a translator needs to know the size).
- For example, in C we can define array types as follows:

```
typedef int TenIntArray [10];
typedef int IntArray [];
```

and we can now define variables as follows:

```
TenIntArray x;
int y[5];
int z[] = {1,2,3,4};
IntArray w = {1,2};
```

- Note that each of these variables has its size determined statically, either by the array type itself or by the initial value list: x has size 10; y, 5; z, 4; and w, 2.
- Also, it is illegal to define a variable of type IntArray without giving an initial value list:

```
IntArray w; /* illegal C! */
```

- Indeed, in C the size of an array cannot even be a computed constant—it must be a literal:

```
const int Size = 5;
int x[Size]; /* illegal C, ok in C++ */
int x[Size*Size] /* illegal C, ok in C++ */
```

- And, of course, any attempt to dynamically define an array size is illegal (in both C and C++)

```
int f(int size) {
int a[size]; /* illegal */
. . .
}
```

- C does allow arrays without specified size to be parameters to functions (these parameters are essentially pointers):

```
int array_max (int a[], int size){
int temp, i;
temp = a[0];
for (i = 1; i < size; i++)
if (a[i] > temp) temp = a[i];
return temp;
}
```

- General function and procedure types can also be created in some languages.
- For example, in C we can define a function type from integers to integers as follows:

```
typedef int (*IntFunction)(int);
```

and we can use this type to define either variables or parameters:

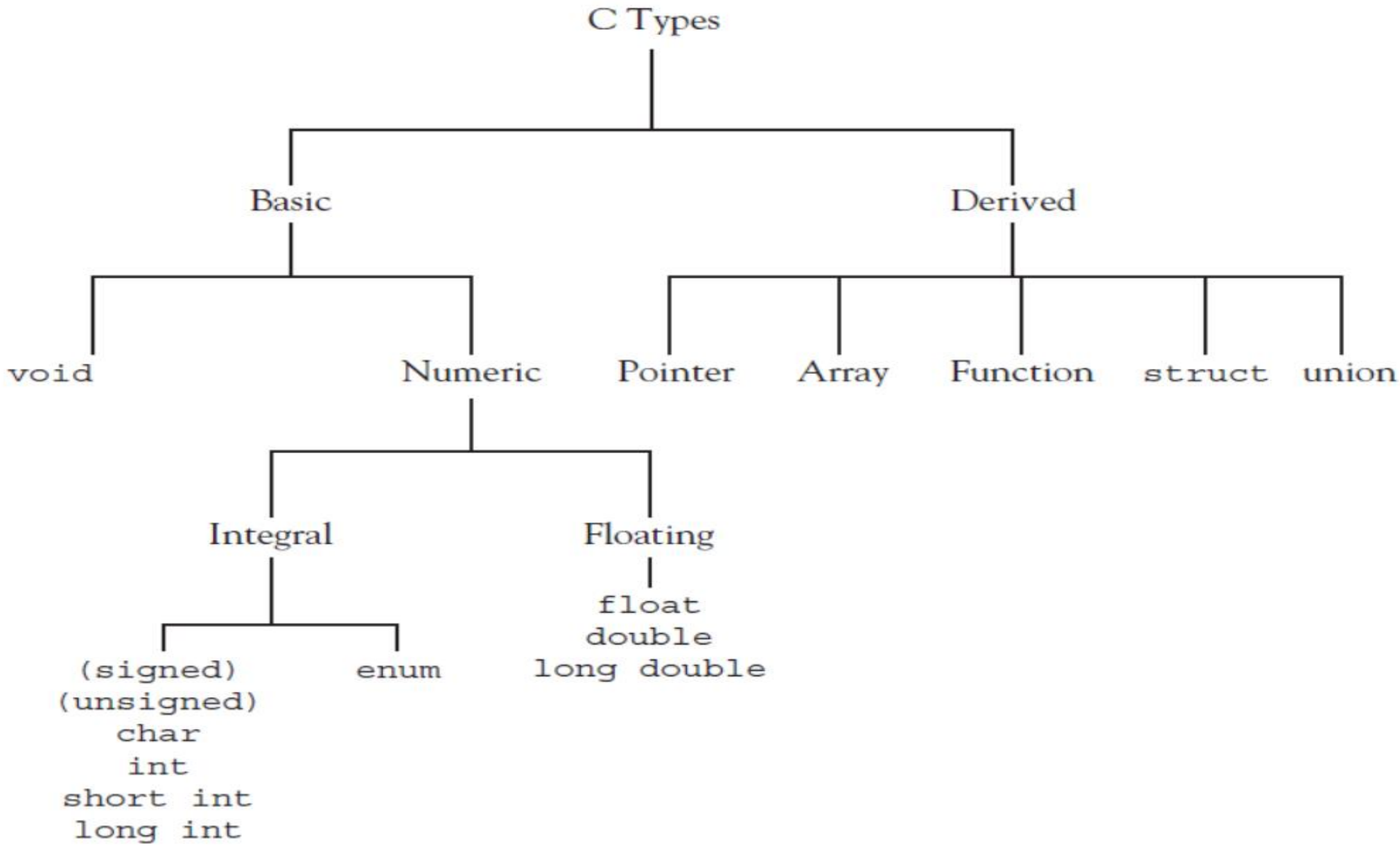
```
int square(int x) { return x*x; }
IntFunction f = square;
int evaluate(IntFunction g, int value)
{ return g(value); }
. . .
printf("%d\n", evaluate(f,3)); /* prints 9 */
```

Type Nomenclature in Sample Languages

- Various language definitions use different and confusing terminology to define similar things.

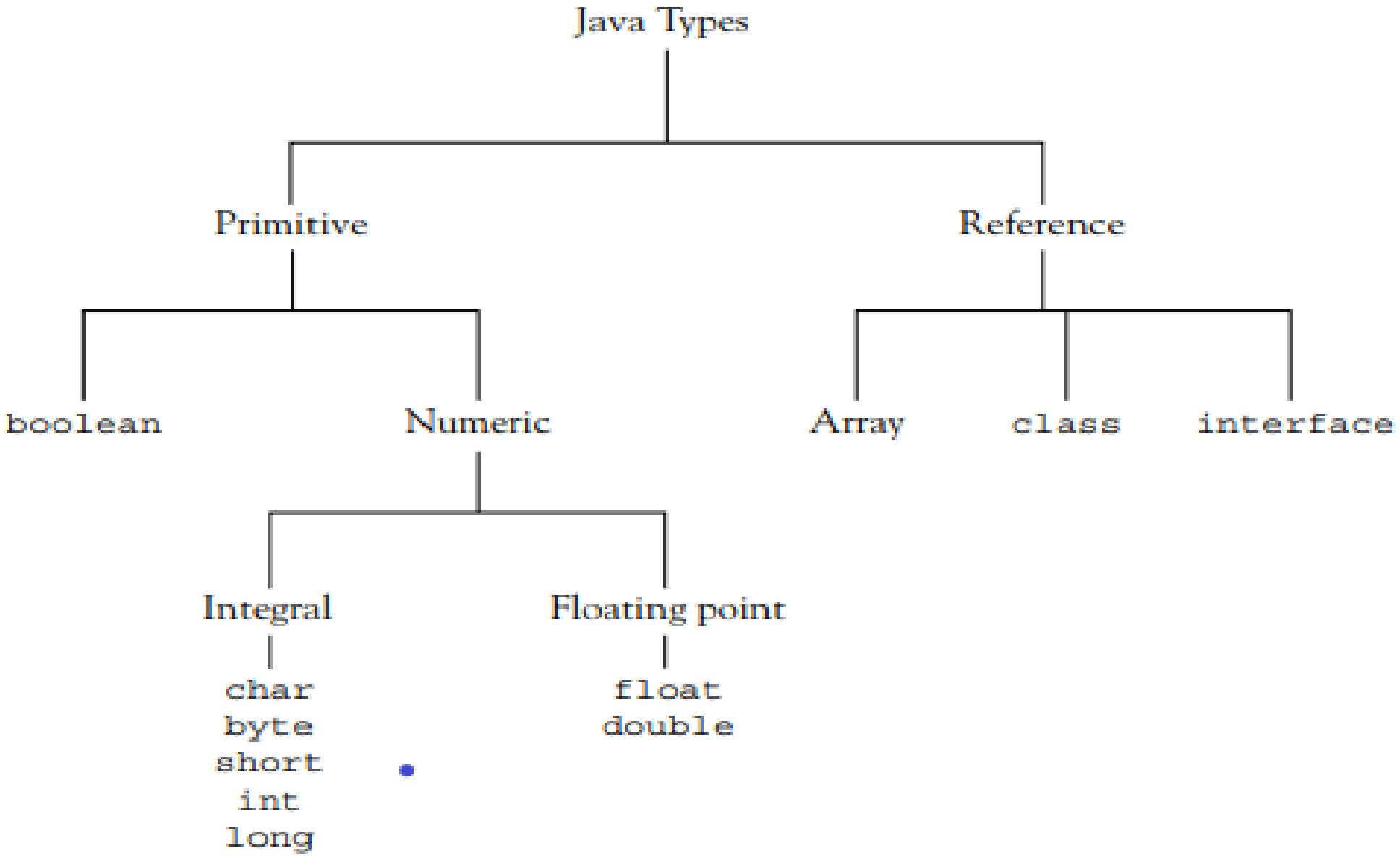
1. C language

- The simple types are called basic types in C, and types that are constructed using type constructors are called derived types.
- The basic types include: the void type (in a sense the simplest of all types, whose set of values is empty); the numeric types; the integral types, which are ordinal; and the floating types.
- There are three kinds of floating types and 12 possible kinds of integral types.
- Among these there are four basic kinds, all of which can also have either signed or unsigned attributes given them


C Types
Basic
Derived
void
Numeric
Pointer
Array
Function
struct
union
Integral
Floating
(signed)
(unsigned)
char
int
short int
long int
enum
float
double
long double

2. Java

- In Java the simple types are called primitive types, and types that are constructed using type constructors are called reference types (since they are all implicitly pointers or references).
- The primitive types are divided into the single type boolean (which is not a numeric or ordinal type) and the numeric types, which split as in C into the integral (ordinal) and floating-point types (five integral and two floating point).
- There are only three type constructors in Java: the array (with no keyword as in C), the class, and the interface.


Java Types
Primitive
Reference
boolean
Numeric
Array
class
interface
Integral
Floating point
char
byte
short
int
long
float
double

Type Equivalence

- A major question involved in the application of types to type checking has to do with type equivalence: When are two types the same? One way of trying to answer this question is to compare the sets of values simply as sets.
- Two sets are the same if they contain the same values: For example, any type defined as the Cartesian product A X B is the same as any other type defined in the same way.
- On the other hand, if we assume that type B is not the same as type A, then a type defined as A X B is not the same as a type defined as B X A, since A X B contains the pairs (a, b) but B X A contains the pairs (b, a).
- This view of type equivalence is that two data types are the same if they have the same structure: They are built in exactly the same way using the same type constructors from the same simple types.
- This form of type equivalence is called structural equivalence and is one of the principal forms of type equivalence in programming languages.

Example

In a C-like syntax, the types struct Rec1 and struct Rec2 defined as follows are structurally equivalent, but struct Rec1 and struct Rec3 are not (the char and int fields are reversed in the definition of struct Rec3):

```
struct Rec1{
char x;
int y;
char z[10];
};
```

```
struct Rec2{
char x;
int y;
char z[10];
};
```

```
struct Rec3{
int y;
char x;
char z[10];
};
```

If structures are taken to be just Cartesian products, then, for example, the two structures

```
struct RecA{
char x;
int y;
};
```

and

```
struct RecB{
char a;
int b;
};
```

should be structurally equivalent; typically, however, they are not equivalent, because variables of the different structures would have to use different names to access the member data.

- A complicating factor is the use of type names in declarations.
- As noted previously, type expressions in declarations may or may not be given explicit names.
- For example, in C, variable declarations may be given using anonymous types (type constructors applied without giving them names), but names can also be given right in structs and unions, or by using a typedef. For example, consider the C code:

```
struct RecA{
char x;
int y;
} a;
typedef struct RecA RecA;
```

```
typedef struct{
char x;
int y;
} RecB;
RecB b;
```

```
struct{
char x;
int y;
} c;
```

- Variable a has a data type with two names: struct RecA and RecA (as given by the typedef).
- Variable b's type has only the name RecB (the struct name was left blank).
- And variable c's type has no name at all! (Actually, c's type still has a name, but it is internal and cannot be referred to by the programmer).
- Of course, the types struct RecA, RecA, RecB, and c's anonymous type are all structurally equivalent.

- Structural equivalence in the presence of type names remains relatively simple—simply replace each name by its associated type expression in its declaration—except for recursive types, where this rule would lead to an infinite loop.
- Consider the following example (a variation on an example used previously):

```
typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;
struct CharListNode{
char data;
CharList next;
};
struct CharListNode2{
char data;
CharList2 next;
};
```

- Clearly CharList and CharList2 are structurally equivalent, but a type checker that simply replaces type names by definitions will get into an infinite loop trying to verify it!
- The secret is to assume that CharList and CharList2 are structurally equivalent to start with.
- It then follows easily that CharListNode and CharListNode2 are structurally equivalent, and then that CharList and CharList2 are indeed themselves equivalent.
- Of course, this seems like a circular argument, and it must be done carefully to give correct results.
- To avoid this problem, a different, much stricter, type equivalence algorithm was developed that focuses on the type names themselves.
- Two types are the same only if they have the same name.
- For obvious reasons, this is called name equivalence.

Example

In the following C declarations,

```
struct RecA{
char x;
int y;
};
typedef struct RecA RecA;
struct RecA a;
RecA b;
struct RecA c;
struct{
char x;
int y;
} d;
```

all of the variables a, b, c, and d are structurally equivalent. However, a and c are name equivalent (and not name equivalent to b or d), while b and d are not name equivalent to any other variable.

Similarly,

given the declarations

typedef int Ar1[10];

typedef Ar1 Ar2;

typedef int Age;

types Ar1 and Ar2 are structurally equivalent but not name equivalent, and Age and int are structurally equivalent but not name equivalent.

- Name equivalence in its purest form is even easier to implement than structural equivalence, as long as we force every type to have an explicit name (this can actually be a good idea, since it documents every type with an explicit name):
- Two types are the same only if they are the same name, and two variables are type equivalent only if their declarations use exactly the same type name.
- We can also invent aliases for types (e.g., Age above is an alias for int), and the type checker forces the programmer to keep uses of aliases distinct (this is also a very good design tool).

- The situation becomes slightly more complex if we allow variable or function declarations to contain new types (i.e., type constructors) rather than existing type names only.
- Consider, for example, the following declarations:

```
struct{
char x;
int y;
} d,e;
```

- Are d and e name equivalent? Here there are no visible type names from which to form a conclusion, so a name equivalence algorithm could say either yes or no.

- A language might include a rule that a combined declaration as this is equivalent to separate declarations:

```
struct{
char x;
int y;
} d;
struct{
char x;
int y;
} e;
```

- In this case, new internal names are generated for each new struct, so d and e are clearly not name equivalent.
- On the other hand, using a single struct in a combined declaration could be viewed as constructing only one internal name, in which case d and e are equivalent.

- C uses a form of type equivalence that falls between name and structural equivalence, and which can be loosely described as “name equivalence for structs and unions, structural equivalence for everything else.”

Type Checking

- Type checking, as explained at the beginning of the chapter, is the process by which a translator verifies that all constructs in a program make sense in terms of the types of its constants, variables, procedures,
- and other entities.
- It involves the application of a type equivalence algorithm to expressions and statements, with the type-checking algorithm varying the use of the type equivalence algorithm to suit the context.
- Thus, a strict type equivalence algorithm such as name equivalence could be relaxed by the type checker if the situation warrants.
- Type checking can be divided into **dynamic and static checking**.
- If type information is maintained and checked at runtime, the checking is dynamic. Interpreters by definition perform dynamic type checking.

- The alternative to dynamic typing is static typing: The types of expressions and objects are determined from the text of the program, and type checking is performed by the translator before execution.
- In a strongly typed language, all type errors must be caught before runtime, so these languages must be statically typed, and type errors are reported as compilation error messages that prevent execution.
- However, a language definition may leave unspecified whether dynamic or static typing is to be used.

- Example
- C compilers apply static type checking during translation, but C is not really strongly typed since many type inconsistencies do not cause compilation errors but are automatically removed through the generation
- of conversion code, either with or without a warning message.
- Most modern compilers, however have error level settings that do provide stronger typing if it is desired.
- C++ also adds stronger type checking to C, but also mainly in the form of compiler warnings rather than errors (for compatibility with C).
- Thus, in C++ (and to a certain extent also in C), many type errors appear only as warnings and do not prevent execution.

- An essential part of type checking is type inference, where the types of expressions are inferred from the types of their subexpressions.
- Type-checking rules (that is, when constructs are type correct) and type inference rules are often intermingled.
- For example, an expression e1 + e2 might be declared type correct if e1 and e2 have the same type, and that type has a “1” operation (type checking), and the result type of the expression is the type of e1 and e2 (type inference).
- This is the rule in Ada, for example.
- In other languages this rule may be softened to include cases where the type of one subexpression is automatically convertible to the type of the other expression.
- As another example of a type-checking rule, in a function call, the types of the actual parameters or arguments must match the types of the formal parameters (type checking), and the result type of the call is the result type of the function (type inference).

- The process of type inference and type checking in statically typed languages is aided by explicit declarations of the types of variables, functions, and other objects.
- For example, if x and y are variables, the correctness and type of the expression x + y is difficult to determine prior to execution unless the types of x and y have been explicitly stated in a declaration.
- However, explicit type declarations are not an absolute requirement for static typing: The languages ML and Haskell perform static type checking but do not require types to be declared.
- Instead, types are inferred from context using an inference mechanism that is more powerful.

Type Compatibility

- Sometimes it is useful to relax type correctness rules so that the types of components need not be precisely the same according to the type equivalence algorithm.
- For example, we noted earlier that the expression e1 + e2 may still make sense even if the types of e1 and e2 are different.
- In such a situation, two different types that still may be correct when combined in certain ways are often called **compatible types**.
- In C and Java, all numeric types are compatible (and conversions are performed such that as much information as possible is retained).

Type Conversion

- In every programming language, it is necessary to convert one type to another under certain circumstances.
- Such type conversion can be built into the type system so that conversions are performed automatically.
- For example, in the following C code:

```
int x = 3;
. . .
x = 2.3 + x / 2;
```

- at the end of this code x still has the value 3: x / 2 is integer division, with result 1, then 1 is converted to the double value 1.0 and added to 2.3 to obtain 3.3, and then 3.3 is truncated to 3 when it is assigned to x.

- In this example, two automatic, or implicit conversions were inserted by the translator.
- The first is the conversion of the int result of the division to double (1 → 1.0) before the addition .
- The second is the conversion of the double result of the addition to an int (3.3 → 3) before the assignment to x.
- Such implicit conversions are sometimes also called coercions.
- The conversion from int to double is an example of a widening conversion, where the target data type can hold all of the information being converted without loss of data, while the conversion from double to int is a narrowing conversion that may involve a loss of data.

- An alternative to implicit conversion is explicit conversion, in which conversion directives are written right into the code.
- Such conversion code is typically called a cast and is commonly written in one of two varieties of syntax.
- The first variety is used in C and Java and consists of writing the desired result type inside parentheses before the expression to be converted.
- Thus, in C we can write the previous implicit conversion example as explicit casts in the following form:

```
x = (int) (2.3 + (double) (x / 2));
```

- The other variety of cast syntax is to use function call syntax, with the result type used as the function name and the value to be converted as the parameter argument.
- C++ and Ada use this form. Thus, the preceding conversions would be written in C++ as:

```
x = int( 2.3 + double( x / 2 ) );
```

- An alternative to implicit conversion is explicit conversion, in which conversion directives are written right into the code.
- Such conversion code is typically called a cast and is commonly written in one of two varieties of syntax.
- The first variety is used in C and Java and consists of writing the desired result type inside parentheses before the expression to be converted.
- Thus, in C we can write the previous implicit conversion example as explicit casts in the following form:

```
x = (int) (2.3 + (double) (x / 2));
```

- The other variety of cast syntax is to use function call syntax, with the result type used as the function name and the value to be converted as the parameter argument.
- C++ and Ada use this form. Thus, the preceding conversions would be written in C++ as:

```
x = int( 2.3 + double( x / 2 ) );
```

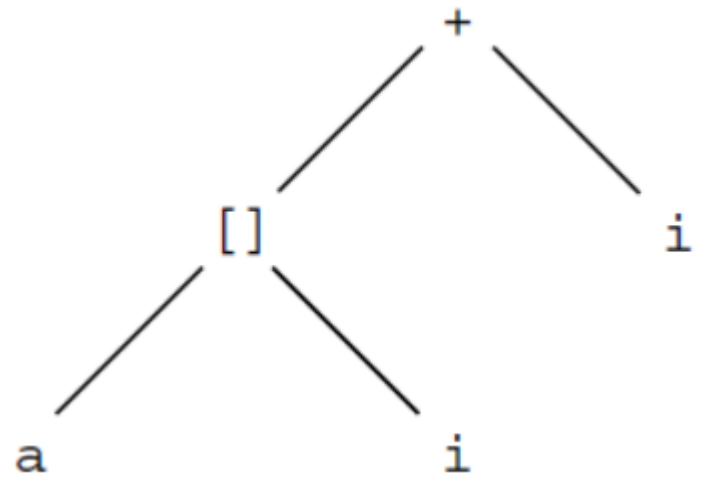
- The advantasge to using casts is that the conversions being performed are documented precisely in the code, with less likelihood of unexpected behavior, or of human readers misunderstanding the code.
- For this reason, a few languages prohibit implicit conversions altogether, forcing programmers to write out casts in all cases. Ada is such a language.
- Additionally, eliminating implicit conversions makes it easier for the translator (and the reader) to resolve overloading.
- For example, in C++ if we have two function declarations:

  double max (int, double);

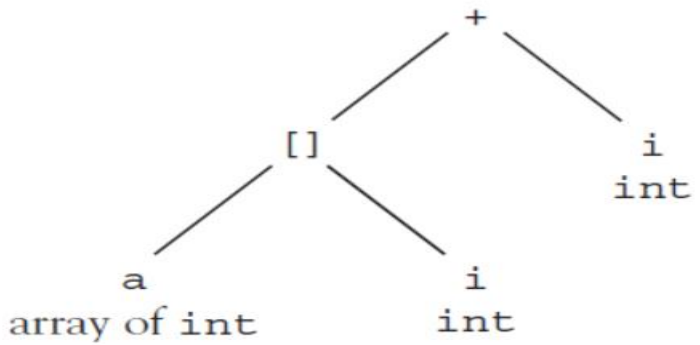
  double max (double,int);
- then the call max(2,3) is ambiguous because of the possible implicit conversions from int to double (which could be done either on the first or second parameter).
- If implicit conversions were not allowed, then the programmer would be forced to specify which conversion is meant.

Polymorphic Type Checking

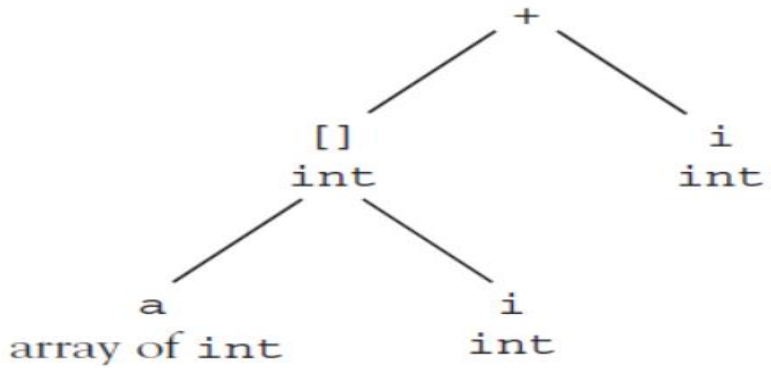
- First, consider how a conventional type checker works.
- For purposes of discussion, consider the expression (in C syntax) a[i] + i.
- For a normal type checker to work, both a and i must be declared, a as an array of integers, and i as an integer, and then the result of the expression is an integer.
- Syntax trees show this in more detail. The type checker starts out with a tree



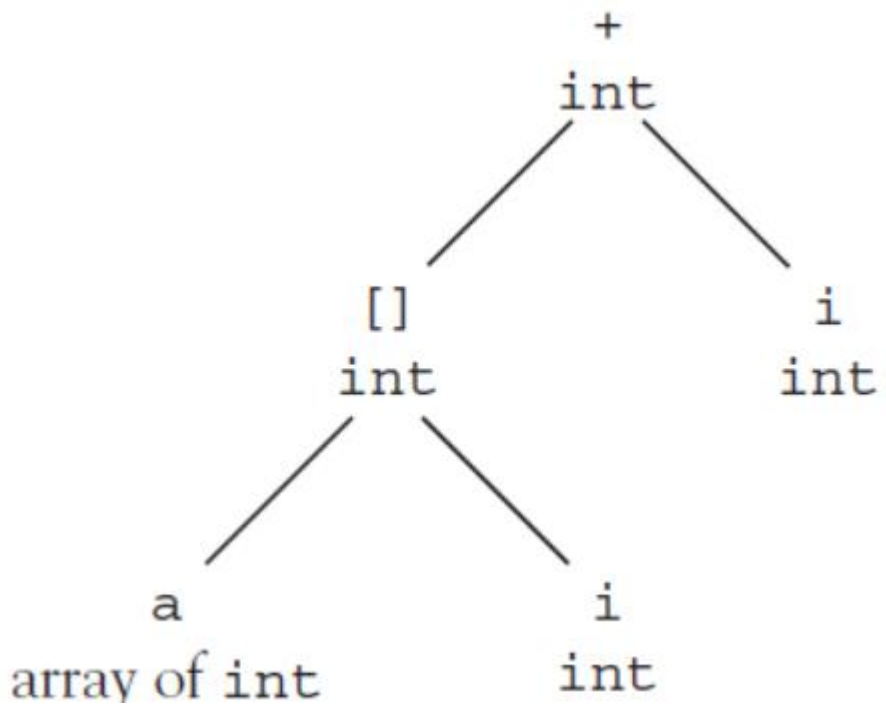
- First, the types of the names (the leaf nodes) are filled in from the declarations



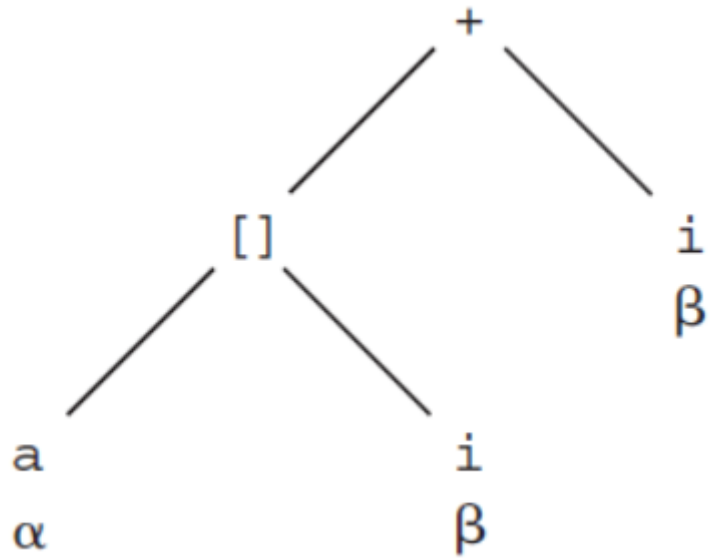
- The type checker then checks the subscript node (labeled []); the left operand must be an array, and the right operand must be an int; indeed, they are, so the operation type checks correctly.
- Then the inferred type of the subscript node is the component type of the array, which is int, so this type is added to the tree



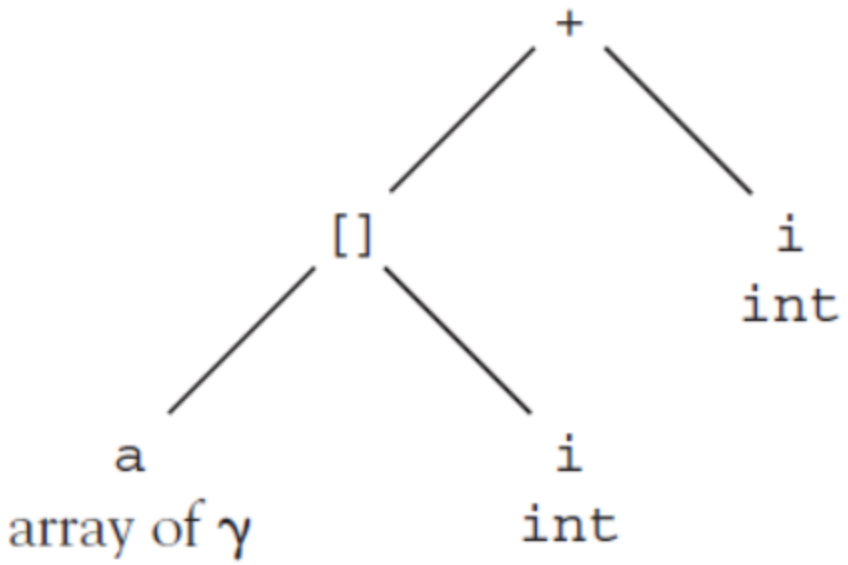
- Finally, the + node is type checked; the two operands must have the same type (we assume no implicit conversions here), and this type must have a + operation.
- Indeed, they are both int, so the + operation is type correct.
- Thus, the result is the type of the operands, which is int.



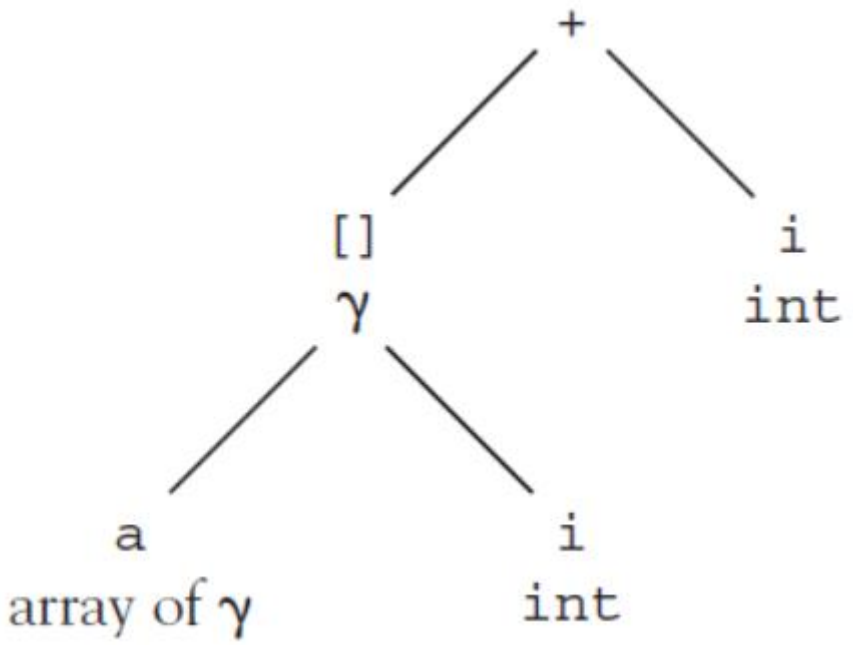
- Could the type checker have come to any other conclusion about the types of the five nodes in this tree? No, not even if the declarations of a and i were missing. Here's how it would do it.
- First, the type checker would assign type variables to all the names for which it did not already have types.
- Thus, if a and i do not yet have types, we need to assign some type variable names to them.
- Since these are internal names that should not be confused with program identifiers, let us use a typical convention and assign the Greek letters α and β as the type variables for a and i.



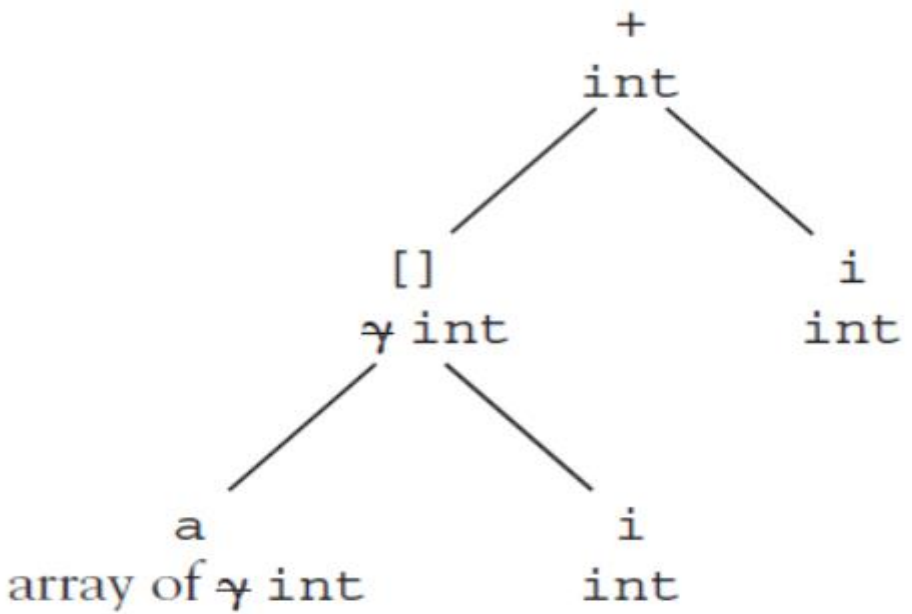
- Now the type checker visits the subscript node, and infers that, for this to be correct, the type of a must actually be an array (in fact, array of ɣ, where ɣ is another type variable, since we don't yet have any information about the component type of a).
- Also, the type checker infers that the type of i must be int (we assume that the language only allows int subscripts in this example). Thus, β is replaced by int in the entire tree, and the situation looks like



- Finally, the type checker concludes that, with these assignments to the type variables, the subscript node is type correct and has the type ɣ (the component type of a).



- Now the type checker visits the + node, and concludes that, for it to be type correct, ɣ must be int (the two operands of + must have the same type). Thus g is replaced by int everywhere it appears, and the result of the + operation is also int.



- This is the basic form of operation of Hindley-Milner type checking

- We note two things about this form of type checking.
- First, as we have indicated, once a type variable is replaced by an actual type (or a more specialized form of a type), then all instances of that variable must be updated to that same new value for the type variable. This process is called instantiation of type variables, and it can be achieved using various forms of indirection into a table of type expressions that is very similar to a symbol table.
- The second thing to note is that when new type information becomes available, the type expressions for variables can change form in various ways. For example, in the previous example a became “array of ɣ,” β became int, and then ɣ became int as well. This process can become even more complex. For example, we might have two type expressions “array of α” and “array of β” that need to be the same for type checking to succeed. In that case, we must have α == β, and so β must be changed to α everywhere it occurs (or vice versa). This process is called unification;

- Consider, for example, the following expression (again in C syntax):

  a[i] = b[i]

- This assigns the value of b[i] to a[i] (and returns that value). Suppose again that we know nothing in advance about the types of a, b, and i. Hindley-Milner type checking will then establish that i must be an int, a must be an "array of α," and b must be an "array of β," and then (because of the assignment, and assuming no implicit conversions), α == β. Thus, type checking concludes with the types of a and b narrowed to "array of α," but α is still a completely unconstrained type variable that could be any type.
- This expression is said to be **polymorphic**, and Hindley-Milner type checking implicitly implements **polymorphic type checking**—that is, we get such polymorphic type checking "for free" as an automatic consequence of the implicit type variables introduced by the algorithm.

- The type “array of α” is actually a set of infinitely many types, depending on the (infinitely many) possible instantiations of the type variable α.
- This type of polymorphism is called parametric polymorphism because a is essentially a type parameter that can be replaced by any type expression.
- In fact, we have seen only implicit parametric polymorphism, since the type parameters (represented by Hindley-Milner type variables) are implicitly introduced by the type checker, rather than being explicitly written by the programmer.

Explicit Polymorphism

- Implicit parametric polymorphism is fine for defining polymorphic functions, but it doesn't help us if we want to define polymorphic data structures.
- Suppose, for example, that we want to define a stack that can contain any kind of data (here implemented as a linked list):

```
typedef struct StackNode{
?? data;
struct StackNode * next;
} * Stack;
```

- Here the double question mark stands for the type of the data, which must be written in for this to be a legal declaration.
- Thus, to define such a data type, it is impossible to make the polymorphic type implicit; instead, we must write the type variable explicitly using appropriate syntax.
- This is called explicit parametric polymorphism.

- Explicitly parameterized polymorphic data types fit nicely into languages with Hindley- Milner polymorphic type checking, but in fact they are nothing more than a mechanism for creating user-defined type constructors.