Chapter 15

The Semantics of Types

Overview

In this chapter we explore the semantics of types. In addition to describing physical properties of objects, types can be used to define the domain of a function and thereby control its application. There are several different approaches to domains and domain or type checking.

We consider the use of types and the development of type checking from the earliest languages to modern strongly-typed languages. Types are properties of data objects and they describe data size, structure, and encoding. Types may also be associated with identifiers to restrict the domain of objects to which an identifier may be bound. Type restrictions can be enforced at compile time or at run time.

In early languages, types were used primarily for storage allocation, storage access, and to control application of predefined functions. Two types were considered compatible if they described the same storage format. This definition of compatibility became inadequate when newer languages began using types to carry semantic information. The translators for the newer strongly-typed languages use domain or type checking to ensure the semantic validity of function calls. A major research problem has been to find semantically meaningful extensions of the type compatibility rules.

A distinction is made between external domains, those within a programmer's application area, and internal domains, the semantic groupings of objects or types identified by the language translator. While older languages had a fixed number of distinct domains, type constructors in modern languages permit programmer-defined domains.

Type casts cause a change in the semantics of an object—the domain label is changed, but the bits are not. A conversion is a change in the physical properties of an object—

size, encoding, or reference level. A coercion is a conversion or cast that is automatically invoked by the translator.

While type checking aids the programmer, it creates an inflexible environment. A programmer who needs to convert from integer to floating point or to compute a hash index, for example, would be hampered by strong type checking. Modern languages often provide escape hatches so that the experienced programmer can evade the type matching rules.

15.1 Semantic Description

Types can be used to embody both the physical properties and the semantic properties of objects. In Chapter 5, we explored the use of types to describe physical properties. In this chapter we look at the other use of types: to define the domain of a function and thereby to control function applicability. A *domain* is the set of objects over which a function is defined. Objects in this set must share common physical properties (size, structure) and semantic properties (encoding, intent).

Functions defined for a domain depend on the common properties of its elements. If the size or structure of a function parameter differed from what the function expected, the results of the computation would probably be wrong. Similarly you would get nonsense if the meaning of an actual parameter was different from the meaning of objects for which the function was designed. Data type definitions were developed as a way to specify the size and structure of variables, so that translators could allocate appropriate amounts of storage. Type checking is a way the translator can use the same information to help the programmer eliminate errors and inconsistencies in code.

Checking was minimal in early languages, but it has become more sophisticated through the years. In this section we look briefly at several different approaches to domains and domain checking. We consider the typing rules in a series of languages, from very old to fairly new.

15.1.1 Domains in Early Languages

Assemblers, the earliest computer languages, had implicit domains rather than explicit domains: addresses, integers, indices, and the like [Exhibit 15.1]. The programmer used elements of these domains, but their properties and relationships were part of the programming lore, not part of the language. All assembly language objects were represented by storage locations (or blocks of locations), each location big enough to store an integer. The type of the object (integer, address, or index) was not part of the program—it existed only in the programmer's mind. Computers typically had some instructions intended to do useful things on each domain. However, the language translator had no way of knowing whether a variable represented a true integer, an address, or an index. Thus the translator could not ensure meaningful use of instructions. Some higher-level

Exhibit 15.1. Operations defined for assembly language domains.

Domain Name	Operations Defined
machine address	Goto, fetch a value, store a value.
integer	Arithmetic and comparison operations.
index	Load index register, add to base address of an array.

languages, such as FORTH, also use a single domain to represent integers, addresses, and indices, and provide no way to ensure their appropriate use.

Early Typed Languages. Languages designed in the 1950s, such as FORTRAN and ALGOL, embodied fixed sets of *primitive*, or predefined, domains. The language and the translator made distinctions among these domains. The domain of each variable was defined (by default in FORTRAN, by declaration in ALGOL) and became permanently associated with the variable identifier.

The original FORTRAN was a very primitive language. It did not have variable declarations; the type of a variable was derived by default from the first letter of its name. Names were restricted to six letters; this is so short that meaningful names were hard to devise. The domains "integer" and "real" were supported as unrelated numeric types that could not be used in combination.

The ALGOL domain structure was much richer than FORTRAN. Integers and reals became related domains [Exhibit 15.2]. Automatic type conversions were introduced so that values of either numeric type could be mixed in arithmetic statements. The external domain "real" was commonly represented internally by single machine words in floating-point encoding.[1] Integers were also commonly represented as single machine words in binary sign and magnitude encoding. With this representation, the internal domain "integer" is not a strict subset of the internal domain "floating point". Some numbers can be represented exactly both ways, some cannot. Very large integers have too many digits of accuracy and can only be approximated in floating-point representation. Nonintegral floating-point numbers can only be approximated in integer representation.

ALGOL introduced `Boolean` as a distinct domain. Boolean values were produced by comparison operators and used by conditional statements. All variables were declared, and the declaration was used for both allocation and type checking. ALGOL had too many primitive types to make FORTRAN-style defaults useful.

15.1.2 Domains in "Typeless" Languages

Some languages, which are called *dynamically typed languages*[2], support dynamic allocation of storage objects whenever they are needed to contain an input value or the result of a computation.[3] Identifiers are *typeless* in the sense that types are not permanently attached to them. Rather, a

[1] These were 36 bits long on the IBM 704.
[2] The less precise term *typeless languages* has also been used.
[3] These languages are usually interpreted, not compiled.

Exhibit 15.2. Domain relationships in ALGOL.

Unrelated. The domains "Boolean" and "integer" were independent. No relationship existed between integer values and Boolean values.

Intersection. The external domain "integer" is a subset of the external domain "real"; all integers are reals, and some reals are integers. However, in a typical 4-byte implementation, the domains integer and real intersect, but neither is a subset of the other.

Numbers Not Exactly Representable as Integers

5,000,000,000	Larger than maximum 4-byte integer.
3.25	Has a fractional part.

Numbers Exactly Representable as Integer and Float

2,147,481,880	No bits in low-order byte of this integer.
3	No bits in high-order byte of this integer.

Number Not Exactly Representable as a Float

2,101,111,111	Integer has "1" bits in both high-and low-order bytes.

Number Exactly Representable as Neither

1/3	Has fractional part that is infinite when expressed in binary.

type tag is attached to each storage object when it is created. The storage object is then bound dynamically to an identifier. Thus the type that is indirectly associated with an identifier may change dynamically. Examples of such languages are LISP, APL, and SNOBOL.

In these languages, the programmer does not *declare* types for variables and function parameters. However, all those variables *have* types, which are necessary to describe the data size, structure, and encoding. These languages all incorporate the various data encodings supported by typical computer instruction sets (character, integer, floating point, bitstring). Further, the domain of an object is usually tested, at run time, before a primitive operation is applied to it. Exhibit 15.3 gives examples of domain-checking in APL, a dynamically typed language. A run-time error comment is generated if the domain is not appropriate.

Types are not used to control function applicability in these languages. Any *programmer-defined* function can be applied to any object. The language does not support the concept that a function might be meaningful for some arguments but not for others. However, even though the domains of arguments to programmer-defined functions are not checked, many (but not all) semantic errors are detected when the program eventually calls a primitive operation with an inappropriate parameter.

In Exhibit 15.4 we define a simple APL function named `DEMO` that accepts a parameter, `S`, passed by value. At function exit, the value of `R` will be returned as the value of the function. Meaningful and meaningless calls on `DEMO` (and their results) are shown in Exhibit 15.5. The domain mismatch

Exhibit 15.3. Domains are checked by APL primitive operators.

The domain of an argument is checked by primitive operators. Use of an operand belonging to the wrong domain results in an error comment.

Programmer writes	APL's output	Note below
A ← 2.11 34.2 17 18.1		(a)
□ ← A [3]	17	(b)
□ ← A [$'N'$]	Domain Error	(c)

a. An array of four numbers is created and bound to the variable A.

b. "□ ←" means to output the value of the expression on the right side of the arrow. The value of the third element of the array A is selected and printed.

c. A subscript in APL must belong to the domain "integer". A character may not be used as a subscript.

Exhibit 15.4. A simple APL function.

Note that the domain of the programmer-defined function is not declared and therefore cannot be checked.

```
▽ R ← DEMO S
[1] A ← 2.11  34.2  4.8  18.1
[2] R ← A[S]
▽
```

Line 1 We define a length-4 array.

Line 2 The parameter, `S`, is used to subscript `A`. If the subscript is in the range 1..4, the corresponding item in the array will be selected and bound to `R`, the local name for the result.

Exhibit 15.5. APL does not check domains of programmer-defined functions.

Programmer writes	APL's response	Notes
⎕ ←DEMO 3	4.8	(a)
⎕ ←DEMO 'N'	Domain Error, line 2 of DEMO	(b)
⎕ ←DEMO 5	Index error, line 2 of DEMO	(c)

a. A correct function call. A[3] is returned.

b. The domain is checked on a call to a primitive function.

c. The subscript range is also checked. 5 is too large for array A.

in the second call is not detected when control enters `DEMO`. However, it is detected during the subscript computation, because subscript is a primitive function.

Dynamically typed languages often supply domain predicates for the primitive domains, so that programmers may write their own domain checks. The most general kind of domain predicate is a Boolean function of two parameters; let us call it `IN`. One parameter is an object, `Ob`, the other is a domain name, `D`. The predicate `IN(Ob, D)` returns `TRUE` if `D` is the domain of `Ob`.

Some languages do not permit a domain name to be used as a parameter. Such a language might supply a separate domain predicate for each domain. In this case the domain name is made part of the predicate name, and each predicate tests whether its single argument belongs to a specific domain. This is well illustrated by the domain predicates in LISP [Exhibit 15.6].

If a dynamically typed language supports domain predicates, a programmer can do manual domain checking within functions. A function will accept, as arguments, objects of any domain. Within the function the programmer writes conditional statements that test the domain and take one of several branches. Conditionals can be set up that will emulate the checking done automatically in Pascal. Thus, whatever operations are ultimately applied to the data, they are sure to be

Exhibit 15.6. Types and domain checking in LISP.

The information given here is for the dialect Common LISP.

Automatic domain checking: primitive functions check the domains of their arguments.

Primitive domains: number (integer), symbol (the name of an object), atom (any nonlist entity), list (a sequence of lists and/or atoms, delimited by parentheses).

Primitive domain predicates: numberp, symbolp, atom, listp, null (true for empty lists). Example: "`(numberp s)`" returns `T` (for true) if `s` is a number, and `F` otherwise.

appropriate for the data encoding. Domain testing becomes another form of data-validity checking, similar to checking for an absurd data value or a table index that is out of range.

15.1.3 Domains in the 1970s

Early C: Domains Checked for Primitive Operators Only.

Type compatibility is a complex question in C. First, we must distinguish among various versions of C, especially between the old semistandard version of C which is described in the Kernighan and Ritchie book (we refer to this as K&R C) and ANSI C. There are far-reaching differences in the typing and type checking rules between the oldest versions and ANSI C. Here we discuss both K&R C and the ANSI standard.

Second, some, but not all, implementations of K&R C have an accompanying program named LINT, which may be used by the programmer to examine programs and locate Pascal-style type errors. A LINT program defines many things as errors that the accompanying C compiler would let go, and thus LINT defines a language with different semantics than C, but the same syntax. LINT implements type constructors and domain identity checking similar to that in Pascal.

Third, many recent non-ANSI compilers generate nonfatal "warning messages" for Pascal-style type errors, but compile the code anyway. It amounts to a semantic quibble whether the warning messages or the generated code defines the semantics of the language.

Finally, type compatibility for array and record types in K&R C was almost a moot point. There was nothing that you could do coherently with a compound object except store its address into a pointer. Thus the question of type checking for function parameters was reduced to a question of what could be done with pointers.

A rough generalization about early C translators is that types were used primarily for storage allocation and access, as described in Chapter 5. Types were declared so that the compiler could know the number, position, and encoding of the fields of an object. This is essential information for producing object code. The types of arguments to primitive functions and operators were checked. But, as in APL, arguments to programmer-defined functions were not checked at all. A function call could have the wrong number, wrong type, or wrong sequence of arguments, and the violation would not be detected.

Further, the rules for pointers created a gaping hole in the type structure. A pointer, declared to point at one type, could be set to point at an object of a different type. Older C compilers would not comment on this type violation. In newer compilers, this would trigger a warning message, but object code will be generated anyway. In Exhibit 15.7, the pointer `r` is made to point at an object whose type does not match `r`'s declared type. Then a field is selected from `r`'s object, and interpreted as if it were the proper type for `r`, which is untrue. This technique can be used to "fool" the compiler into compiling code that would cause a type error if it were executed without the pointers. Thus the language definition and the compiler provide no sure control over whether or not a pointer is pointing at an object of the appropriate type.

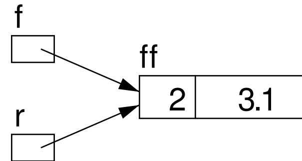
Exhibit 15.7. Pointer assignments in K&R C.

Two types named `funny` and `runny` are defined, and are used to declare two pointers (named `f` and `r`) and an ordinary structured variable (named `ff`). Note that these types are structurally different. The following statements will compile without causing a fatal error. Recent non-ANSI C compilers are likely to produce a warning message for the last assignment.

```
struct runny {int a; int b;}  *r;
struct funny {int a; float c;} *f, ff= {2, 3.1};

f = & ff;    /* This makes f point at the variable ff.  */
r = f;       /* This sets r to point at the same thing as f,
             which is, of course, the wrong type for r.  */
```

After executing these lines, storage will contain:



The first expression, below, is normal. The second field of a `funny` object is selected. It is expected to be, and is, a float. The second expression, though, selects the second field of a `runny` object, which is expected to be an integer, but is actually a float that will be interpreted as an integer.

f–>c   Select 2nd part of f's struct; float result is interpreted as float.
r–>b   Select 2nd part of r's struct; float result is interpreted as int.

Programmer-defined Domains

Time and experience have influenced design philosophy. The historical trend has been to give domains an increasingly important role in programming languages because they are a great aid to producing semantically sound programs. In older languages, all domains were predefined, and their relationships (if any) were predefined and not modifiable. No domain checking was done that distinguished between objects of similarly represented domains. For example, in C (which was developed by Ritchie in about 1972), integers and truth values are the same domain. More important, the types of the arguments to programmer-defined functions were not checked.

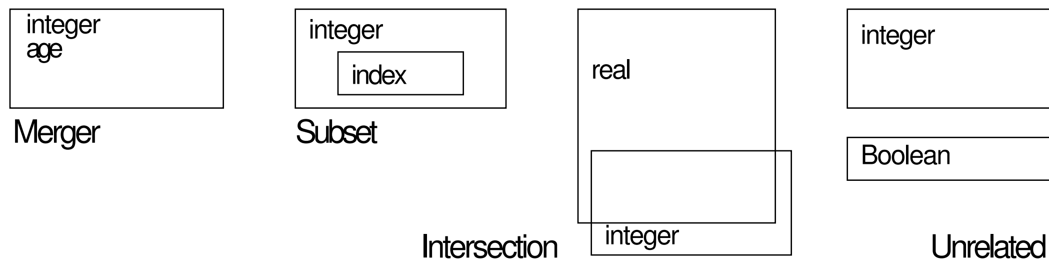
In newer languages, such as Pascal (released by Jensen and Wirth in 1974), ways were provided to define new domains, and more elaborate domain-matching rules were implemented. In addition to the usual record, array, and pointer types, Pascal included several new kinds of type definitions: subrange type, enumerated type,[4] set type, and type mapping [Exhibit 15.8]. Domain relationships began to become important. In Pascal, these were pre-defined and unmodifiable. Let us consider the four types of relationships in Pascal.

[4]Enumerated types were not supported by early versions of C.

Exhibit 15.8. Diagrams of the domain relationships in Pascal.

```
TYPE index = 0..100;
     age   = integer;
```

- Boolean, integer, and real are primitive types.
- The domain Boolean is semantically unrelated to integer, even though it is defined by mapping onto a subset of the integers.
- Integer and real intersect; that is, some reals can be converted to integers; most or all integers (depending on the implementation) are also representable as reals.
- The domain index is a subset of the domain integer, and fully compatible.
- The domain age is merged with the domain integer. There is no distinction.



Intersection. The domains `real` and `integer` intersect in Pascal as they did in ALGOL. Set domains can intersect with other set domains over the same base type.

Unrelated. Domains created by array or record or pointer type declarations are unrelated to any other domain. Domains defined by enumeration (including the primitive domains `char` and `Boolean`) are unrelated semantically to any other domain. However, all enumerated types are represented by integers and can be converted to the domain `integer` by explicit use of the primitive function `ord`.

Subset. The Pascal subrange declaration narrows a domain to a compatible smaller domain with the same representation. Variables of a subrange type are restricted to storing values in the subrange. Attempting to store a value outside this range results in a run-time error and program termination. Using a subrange variable incurs run-time overhead due to automatic range-checking every time a value is stored in it! While the protection provided by subranges is valuable, especially

during program debugging, the cost is great. Further, because a range error causes a Pascal program to "bomb", subranges cannot be used for ordinary input validation. They are primarily useful for trapping subscript values that have unintentionally become too large or too small.

Merger. Compatibility rules were very loose in the original K&R C; they are far tighter in Pascal. For example, truth values, integers, and addresses comprise different domains, and a pointer to a real is incompatible with a pointer to an integer. The compiler checks the domains of all function arguments and requires that the formal and actual parameters have the same domain identity (not just the same representation, as in C).

However, even in Pascal, the relationships among types, structures, and domains are not easily stated. In Pascal, we may have merged domains, that is, more than one type name associated with one domain [Exhibit 15.18], and more than one domain associated with one structural description [Exhibit 15.14]. The number of distinct domains is equal neither to the number of names nor to the number of different structures defined.

15.1.4 Domains in the 1980s

Languages newer than Pascal embody a clearer view of the nature and uses of domains. For example, Ada gives the programmer more control over the relationships among new domains (see Section 15.3).

In C, the programmer specifies the physical size of variables by declaring them to be "`long int`", "`short int`", or "`int`" (meaning either one). In contrast, Ada encourages the programmer to declare domains abstractly, by specifying characteristics (range, precision) of the values to be represented, rather than concretely, by specifying the number of machine locations needed. Programs containing abstract specifications are more easily ported to machines with diverse architectures.

The run-time overhead incurred in Pascal for range checking is also incurred in Ada. However, the Ada designers provided a means (the "pragma") to turn this checking off. The programmer can have the dual advantages of automatic checking during debugging and efficiency when a program enters production use.

Finally, there are languages that let the programmer define a domain with more than one representation. Having multiply represented domains is basic to the modern functional languages and the object-oriented languages.[5]

15.2 Type Checking

Translators use *domain checking* (often called *type checking*) to ensure the semantic validity of function calls. In a domain-checked language, the domain of each formal parameter of a function is declared to be a specific or generic data type. When the programmer tries to apply a function to a set of actual parameters, the translator verifies that the type of each actual parameter is appropriate;

[5]These are discussed in Chapter 17.

the type of the actual parameter must be contained in the domain of the corresponding formal parameter. If this is true we say the actual and formal parameters are *type compatible*, or that their types *match*.

If any parameter fails this test, the function call will not be carried out and an error comment will be generated. This process can be done dynamically, rejecting meaningless function calls at run time, or statically, detecting *type errors* at compile time. The actual rules for determining whether two types match can be confusingly complex and vary greatly from language to language. The rest of this chapter discusses and contrasts the compatibility rules for a few languages.

15.2.1 Strong Typing

The term *strongly typed language* means, roughly, a language that checks the semantic validity of all function calls. There is considerable confusion, though, about the exact meaning of this term. In this section we will go through a series of increasingly sophisticated definitions of strong typing.

Some early authors called a language "typed" (abbreviated as ST#1) if the programmer used declarations to specify the types of variables. By this definition, APL and BASIC are not ST#1 and C and Pascal are. This definition classes languages such as BASIC, (where the data type is implicitly declared by the form of the variable name) with languages such as APL which do not associate types with identifiers at all. A better definition (ST#2) is: a language is typed if there is a type associated with each variable name, and only objects of that type can be stored in the variable. APL is not ST#2 and BASIC, C and Pascal are.

It became clear that the important distinction was not whether every object had a declared type, but whether those types were used to support semantic validity. A new name and a third, stricter definition evolved: A *strongly typed language* (ST#3) is one in which the compiler enforces these type compatibility rules:

- All objects (variables, values, and formal parameters) are divided into sets called *types*, usually labeled by type names. Types can be built in or programmer defined. Each object belongs to exactly one type.
- A type may have two or more variants. The variant to which an object belongs can be ascertained by a program at run time. Perhaps it is encoded by a tag field associated with the object.
- A variable can only store values of the same type.
- In a legal function call, the type of each actual argument must match the type of the corresponding formal parameter. The exact definition of compatibility is language-dependent. Roughly, though, two types are usually compatible if they are the same type or are overlapping subsets of the same type.
- If an argument belongs to a type with variants, the programmer must explicitly test which variant is present and write code that handles each variant appropriately.

APL is not ST#3. Neither is the original Kernighan and Ritchie version of C, because calls to user-defined functions are not type-checked. Pascal, Ada, and ANSI C are ST#3 except for some "escape hatches". These are Pascal's nondiscriminated variant record and the analogous union data type in C, as well as ANSI C's continued support of K&R C's original function definition syntax and semantics. BASIC is ST#3, in a rather uninteresting way, because it will never let you apply numeric operators to strings or vice versa.

When using the method of top-down programming, a programmer begins by working with a very general description of a problem and then making several passes over that description. On each pass the programmer specifies more details of both the data representation and the method for computation. In the middle of this process an abstract solution for the problem has been described. The algorithms and data are defined in a rough fashion but nothing is specified fully yet. Finally, after several passes, the programmer arrives at a fully specified algorithm which can be coded.

Languages that are ST#3 offer limited support for top-down programming, primarily because they require the full specification of data representations too early in the process. Functions can only be written to accept parameters of fully specified types. In order to define functions over abstract data types, a language must permit postponement of the time at which representation details need to be pinned down.

15.2.2 Strong Typing and Data Abstraction

An *abstract data type*, or *ADT*, is a combination of one or more abstract domains together with a set of functions that operate on them, and related data items that characterize the domain. An ADT is an abstraction. We can define an abstraction in English by specifying its required properties. For example, we can define a "tire" as a resilient covering for the rim of a wheel. This definition is a generic definition because it includes "tires" of many sizes, shapes, and materials. Similarly, we can use English to define an ADT; to define "stack" we define the semantics of "push" and "pop" when applied to a stack object.

Only part of this information can be expressed explicitly within a programming language: the intent and semantic properties of the ADT must remain implicit. For example, the fact that the `push` and `pop` operations for the stack ADT implement a last-in-first-out accessing pattern is not expressible in any way except a comment. What we can express in a program are abstract functions (function headers without accompanying bodies) and abstract domains (domain names without accompanying data type definitions). For example, Exhibit 15.9 defines the ADT `array_stack` in an Ada-like language. An `array_stack` is an array-based data structure capable of storing multiple data items, together with the functions `Push` and `Pop` that implement a LIFO accessing pattern, the function `Top` that returns a copy of the most recently pushed item, and predicates `Empty` and `Full`.

We can make a specific implementation of an ADT in an ST#3 language by defining a specific data type to represent each abstract domain in the ADT, and a specific function to carry out each abstract function. For example, to make a specific implementation of a stack, we first define a specific data type to represent the stack. Functions "push" and "pop" are then defined that

Exhibit 15.9. Definition of the ADT "array_stack".

We define the abstract data type array_stack by declaring a parameterized generic domain and listing the relevant abstract functions. We use syntax modeled after Ada, but extended by several constructs that are not supported by Ada.

```
ADT array_stack Begin
    ParamType Stack(
        T:type;             -- Type of data objects to be stored.
        L:integer);         -- Maximum number of simultaneous objects.
    Function Top(
        S: in Stack)        -- Stack itself is unchanged.
        return T;           -- Returns copy of most recently pushed data.
    Function Pop(
        S: in out Stack)    -- Top item is removed from S.
        return T;           -- Returns most recently pushed data.
    Function Push(
        item:  in T,        -- Store item at top of stack.
        S: in out Stack)
        return Boolean;     -- Return true if push executed successfully.
    Function Full(
        S: in Stack )       -- Stack itself is unchanged.
        return Boolean;     -- Return TRUE if S.store is full.
    Function Empty(
        S: in Stack )       -- Stack itself is unchanged.
        return Boolean;     -- Return TRUE if stack contains no data.
End ADT array_stack;
```

operate on that type to carry out the semantics defined for the ADT operations. We are able to define a stack of 100 characters in Pascal (to be represented by an array [1..100] of char), or a stack of integers (to be represented by a linked list of integers). However, we cannot define the abstract data type itself in these languages.

This is because, by definition, an ST#3 language has disjoint (nonoverlapping) types.[6] There is no general provision for talking about groups of types or defining a function that would apply to more than one type, or to a partially unspecified type. A function definition in an ST#3 language specifies the types of its arguments, so every ST#3 function depends on a previously defined specific type and cannot be written in a general way to operate on a generic domain. Thus a program in Pascal that uses two kinds of stacks requires two sets of type declarations and two separate but nearly identical definitions of each stack function.

[6]Variant records and unions are inadequate support for generic types.

Many Pascal students have asked why they could not make one set of definitions for a stack and its operations, then have it automatically applied to stacks of any kind of object. The best that can be done in Pascal is to write a set of stack definitions and store them in a source code library. To use the stack code, the source code must be edited to change the type definitions, then compiled. The language does not support compiled modules with type flexibility, and it does not even provide an automated way to do the editing job.

When Ada was developed, a facility was included to automate this tailoring process. A set of definitions called a *generic package* can be defined in Ada, which contains the type declarations and function definitions for an abstract data type such as "stack". These definitions are written in terms of a type parameter. Later, the package must be instantiated with a concrete type during the first phase of compilation. This process generates ordinary program code that is ST#3, and it is then compiled. Generic packages for standard algorithms on commonly useful data structures can be included in libraries. These are called in and instantiated when needed. The need to write and debug new code is minimized. However, the end product is the same as if the programmer had written fully concrete code in the first place. Thus Ada's generic facility adds real convenience but not significant flexibility to the language.

Chapter 17 describes Ada generic packages and examines the support for data abstraction in more modern languages such as Miranda and C++, which permit the user to define generic types and functions over these types. Chapter 17 also presents a fourth definition of "strong typing" which admits automatic type checking for these nonhomogeneous types.

15.3 Domain Identity: Different Domain/ Same Domain?

15.3.1 Internal and External Domains

There is generally not a one-to-one relationship between type names and distinct domains. Most languages use type names to define domain membership, but a translator may implement two type names as the same domain or as distinguishable domains. It is possible in some languages to define two different type names which have identical semantics and, therefore, denote the same domain, and also to define different semantics for structurally identical types, so that they become distinct domains.

We must first distinguish between external domains and internal domains. *External domains* are those that occur in the programmer's application area. *Internal domains* are the semantic groupings of objects or types that are recognized and maintained by a language translator. These are not necessarily the same thing. Two types, A and B, form *distinguishable internal domains* if the translator implements different semantics for them. Operationally that means that some function is defined differently for type A than it is for type B. Two distinguishable domains are *independent* if no function defined for one is applicable to objects from the other, and vice versa. If this property holds in one direction but not in the other, we say the domains are *semiindependent.* If the implementations of two external domains have the same internal semantics, we say they are *internally merged.*

Exhibit 15.10. Merging the domains "truth value" and "integer".

Neither C nor FORTH distinguishes between integers and truth values. There are many situations in which this is convenient. Programmers using C and FORTH commonly use the "tricks" below to shorten their code. (The examples are written in both languages.)

Using a truth value as an integer. Your process compares two input streams, item by item, and gives a TRUE answer if the items are not equal. You wish to count the number of nonmatching item pairs. This is easy: just "add up" the TRUE values. (TRUE is represented by 1 in C and by −1 in FORTH.)

C: difference_count = difference_count + (item1 != item2);
FORTH: item1 item2 = NOT difference_count @ + difference_count !

Using an integer as a truth value. You wish to terminate a loop when the input variable, an integer, equals zero. Easy! Remember that 0 is the representation for FALSE, and write:

C: while (input_variable) { ⟨process⟩ } ;
FORTH: BEGIN input_variable @ WHILE ⟨process⟩ REPEAT

15.3.2 Internally Merged Domains

Some languages maintain no semantic distinction among domains that have the same internal representation. For example, in FORTH and C, integers, truth values, and characters are represented identically. In these languages, an integer *is* a truth value *is* a character, and a truth value or a character *is* an integer. A value belonging to any one of these external domains can be used in a context appropriate for any other one. An integer operation may be applied at will to a truth value or a character. No "conversion" process is needed to go between these domains.

This is one of the very convenient aspects of C: the translator lets the programmer decide whether it is meaningful to use a truth value as if it were an integer, and thus does not prevent the programmer from exploiting an implicit relationship between domains [Exhibit 15.10].

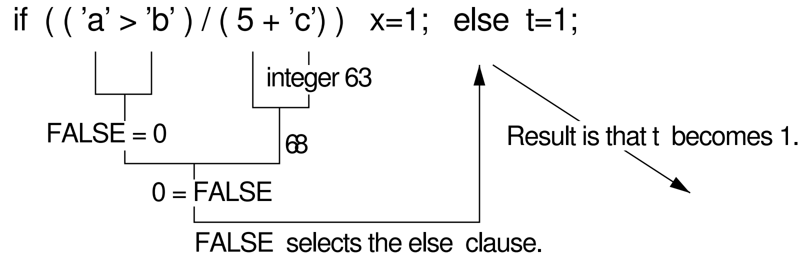
Of course, one could also claim that this kind of code is obscure and ought to be well documented if it is written at all. In fact, C and FORTH programmers use these tricks frequently, and the very commonness of such code reduces the difficulty of understanding it.

A more serious cost associated with internally merged domains is that the translator has no way of knowing which objects belong to which external domain, and so cannot help the programmer avoid unintended and meaningless operations. This is illustrated by the very strange C code in Exhibit 15.11. Let us trace the execution of this odd expression:

- The ASCII codes for `'a'` and `'b'` are compared. `'a'` is not greater than `'b'`, so the answer is FALSE, which is represented by 0 (4 bytes).
- The character `'c'` (one byte) is interpreted as an integer. Its ASCII code, 99, must be lengthened to 4 bytes to match the length of the integer 5.

Exhibit 15.11. Many misinterpretations!

The "/" and the "+" are nonsensical operations. They would be flagged as errors in Pascal, but they are accepted in C because integers, characters, and truth values belong to one internally merged domain.



- We add 5 to 99 giving 104 (4 bytes).
- The truth value, 0, from item (1) is interpreted as an integer and divided by 104, giving 0 (4 bytes).
- This integer 0 is now interpreted as the truth value FALSE, and used in the "if" test, selecting the else clause.
- Finally, we store 1 in the variable t.

This example illustrates the variety of ways that C values can be interpreted, but it is so artificial that it does not provide a convincing example of the cost of internally merged domains. However, this kind of domain merging does frequently lead to errors, especially for programmers who commonly use more than one language. Exhibit 15.12 illustrates a common and galling error that most former Pascal programmers make when they begin to write code in C.

As a general rule, the language that has distinct internal domains representing distinct external domains is easier to learn and easier to use. A compiler with full type checking is a powerful ally in the battle to debug a large program. The price paid for this assistance is that the programmer must explicitly indicate domain conversions.

15.3.3 Domain Mapping

An existing domain may be used to implement a new domain by mapping the elements of the new domain onto the elements of the old. This produces two domains with a common implementation but dissimilar semantic intent. For example, one could represent the imaginary numbers by mapping each imaginary onto the corresponding real and leaving implicit the fact that each one represents that real multiplied by i.

When a new domain, D', is implemented by mapping it onto an old domain, D, we say that every element of D' is *represented by* an element from D. In this case, the object from D' can be

Exhibit 15.12. The bane of the former Pascal programmer.

The Pascal comparison operator is "=", but C uses "==" for comparison and "=" for assignment. Intending to repeat a process many times, as long as the variable named "a" remains equal to zero, the absentminded programmer writes:

```
do ⟨process to be repeated⟩ while ( a=0 );
```

Unfortunately, this process will be executed exactly once. The first time the expression `a=0` is evaluated, the value zero will be stored in the variable `a`, and also returned as the value of the assignment expression. The `while` test interprets the 0 as `FALSE` and terminates the loop.

Moreover, when the perplexed programmer looks at the value in `a`, she or he will see the number zero and be unable to understand *why* the loop did not repeat!

"converted" into an object from D, or vice versa, by changing only the domain identifier attached to the object. The physical form of the value does not need to be changed. The same domain, D, may be used to implement other domains also, making a many-to-one relationship between implemented domains and an implementing domain [Exhibit 15.13].

Although a D-object is structurally identical to a D'-object, different sets of functions are probably appropriate for D and D'. For example, bit strings are sometimes used to represent both integers and arrays of switches. Division is meaningful for integers but not for switches, and masking operations may be meaningful for arrays of switches but not for integers. Concatenation might be a meaningful operation for bit strings that are neither integers nor switches. Thus the semantic intent of the object's domain, not just its physical representation, determines what operations are meaningful. The intent should be considered before applying a function to the object.

Mapped Domains with Distinct Identities

The primitive types in Pascal demonstrate domain mapping where all domains have separate identities. Several Pascal primitive types are actually represented in the computer as integers [Exhibit 15.13]. Pascal also permits the programmer to define an "enumerated type". To do this, the

Exhibit 15.13. Domains mapped onto integers in Pascal.

Semantic Intent	Typical Implementations
integer numbers	full-word bit strings
alphabetic characters	full-word integers (8-bit if packed)
machine addresses	integers 0 .. virtual memory size
truth values (false, true)	integers (0=false, 1=true)
enumerated type	integers 0 .. cardinality of the type

programmer enumerates names for the elements of a finite type, and the type is implemented by mapping the elements, in the order given, onto the integers.

The computer hardware itself defines the mappings for integers, addresses, and characters. Integers are defined by the operation of the machine language add, negate, and comparison instructions. Addresses are mapped onto the integers by the memory mapping hardware of the machine; one computes the next machine address in a sequence by using the integer "add 1" machine instruction. The mapping from characters to bit strings is defined by the I/O devices, most of which implement a common character code such as ASCII.

Truth values are mapped onto bit strings by a language implementor. "False" is virtually always represented as a string of 0 bits. Anything nonzero is therefore taken to be "true". When a "true" value must be generated by the translator, most languages generate the string 00000001 with a number of leading zeros appropriate to fill a memory location. (Some translators generate a string of all 1 bits.)

Pascal defines primitive functions and procedures for these primitive domains. Although the representations of these domains are structurally compatible with each other, the primitive functions only accept arguments from the defined domain. A Pascal translator checks the declared data type of each argument against the declared domain of a function and enforces what is, ideally, the programmer's semantic intent. If the programmer wants to do some operation that would violate the type rules, he or she must first explicitly convert the argument to a different type.

15.4 Programmer-Defined Domains

15.4.1 Type Description versus Type Name

In the historical progression from ALGOL, through C, Pascal, and Ada, to C++ and ML, types have become the basis for increasingly powerful semantic mechanisms and have been given increasingly clear semantics. Languages have supported programmer-defined types since the 1960s. During this time the relation between the type description, the type name, and the type's semantics have varied greatly. In the older languages, such as K&R C, a type name was no more and no less than a shorthand notation for the type description. All types with the same description were merged into one internal domain. In such a language two structurally identical objects belong to the same domain even if they were declared using different type names.

A more modern approach is to give meaning to the type name over and above the meaning of the type description. In such a language the programmer may use the type name to help express semantics. Thus the same type description may be associated with more than one type name to express semantically different types that happen to have the same representation [Exhibit 15.14]. The compiler can then use these differentiated type names to help the programmer achieve semantic validity. In this situation, type checking can catch errors that checking only for structural properties cannot detect.

Exhibit 15.14. Two types with the same description.

In writing a graphics program one might use two related but distinct types:

- A set of points in the Cartesian plane
- A set of points in the polar coordinate plane

Both of these types are normally represented by a pair of real numbers. In the Pascal type declarations below, both are given the same structure. The two type names will not be synonyms, since functions defined for one cannot be applied to the other.

```
TYPE cart_point = ARRAY [1..2] OF real;
     polar_point = ARRAY [1..2] OF real;
```

15.4.2 Type Constructors

Old languages had a fixed number of distinct domains. In modern languages, type constructors are provided to permit the programmer to define new domains. Each new domain may be, and normally is, bound to a new type name. The new type name can be used to declare objects that belong to the domain and to declare the domain of function parameters.

A *type constructor* is a keyword or syntactic construct whose use creates a new domain. Not all ways of declaring new types are type constructors. Type constructors are chosen by a language designer and vary greatly among languages. For example, "`array`" and "↑" are type constructors in Pascal but the analogous "[⟨dimension⟩]" and "*" in C do not construct domains. However, `struct` in C and the analogous `record` in Pascal are both type constructors.

Constructed Domains and Type Checking

When a programmer uses a type constructor in a type declaration, the declared type name is bound to the newly formed domain. Thereafter, other program statements can declare objects and function parameters in that domain by referring to the type name directly or indirectly [Exhibit 15.15].

In an ST#3 language, a domain, D, created by a type constructor is *independent*; that is, it is functionally incompatible with all existing or future domains.[7] No functions may be applied to objects from D unless they are explicitly defined for D, and functions defined for D may not be applied to objects from any other domain. An object is type-compatible with a formal parameter if and only if both were declared to belong to the same domain.

Every occurrence of a type constructor constructs a different domain. If types $D1$ and $D2$ are declarations with identical type definitions containing a type constructor, $D1$ and $D2$ are incompatible. In some languages, a type constructor may be used in a variable or parameter

[7] In more modern languages, this incompatibility may be modified by rules for domain inheritance.

Exhibit 15.15. A type constructor in Pascal.

The keyword array is a type constructor in Pascal; each use of array builds a new domain. Here we construct a new domain and bind the name BoxDimensions to it. We use this type name to define a function, vol, that operates on BoxDimensions.

Giving a new name to a type does not construct a domain in Pascal. The type name Tank names the same domain as BoxDimensions.

```
TYPE  BoxDimensions = array [1..3] of Real;
      Tank = BoxDimensions;
VAR   p, q:  BoxDimensions;
      t:  Tank;

FUNCTION vol(d:  BoxDimensions):  real;
      BEGIN vol := d[1] * d[2] * d[3] END;
```

The function vol may be legally applied to variables p and q. The types match because p, q, and d were all declared to be BoxDimensions. This function may also be applied to t, since it was declared using the same instance of array as was BoxDimensions.

---

declaration. This constructs a domain that is incompatible with everything because the new domain has no name, and other parts of the program are unable to refer to it. The only variables that can ever belong to this domain are those created by the same declaration.

Thus the variables p and q in Exhibit 15.16 have the same (unnamed) domain. Variables r and s have the same domain, structurally identical to the domain of p and q, but semantically distinct because they were declared with different uses of the type constructor array. No other variables or parameters can belong to the same domain as p and q, or as r and s. Specifically, variable v does not belong to the same domain as any of these other variables.

An object and a parameter belonging to different domains are incompatible, even if they have the same structural description. A type constructor, therefore, is never used to declare the parameters of a function: no variable could ever match the type of the parameter, and the function could never be called. In Exhibit 15.16, the function NoGood can never be called because the type of its parameter, t, can never match the type of any variable. Specifically, t is not in the same domain as p, r, or v, but belongs to a fourth domain with distinct semantics.

15.4.3 Types Defined by Mapping

Non-Independent Mapped Types

In some languages, a type defined by mapping does not construct a new, independent domain. Rather, the new type name is an alternative way to refer to an existing domain. The typedef declaration in C is an example. It defines the new type name as a synonym for its definition, which

Exhibit 15.16. Incompatible domains in Pascal.

Four incompatible domains are constructed here, by using the type constructor `array` four times. Each of the identifiers `p`, `r`, `v`, and `t` belongs to a different domain. The function `NoGood` cannot ever be called because its parameter, `t`, is not in the same domain as any other object.

```
TYPE  VitalStats = array [1..3] of real;

VAR   p, q: array [1..3] of real;
      r, s: array [1..3] of real;
      v :   VitalStats;

FUNCTION NoGood(t:  array[1..3] of real):real;
      BEGIN NoGood := t[1] + t[2] + t[3] END;
```

is often a structural description [Exhibit 15.17]. Although `typedef` does not create a new domain, a `typedef` declaration is useful in C because the syntax for using `typedef` names is clearer and more convenient than the syntax for using their descriptions.

C was one of the earliest languages developed that permitted the programmer to declare new types. At that time, the relationships among the name of a type, its representation, and its semantics were only partially understood. This is probably why C has fewer type constructors than newer languages, and why `typedef` is not a type constructor.[8]

[8]The `typedef` declaration was added to C after the language had been in use for some years.

Exhibit 15.17. Merged domains defined using `typedef` in C.

A point in a plane can be represented as a pair of real numbers. We define a record type using the C type constructor `struct`. The identifier `pt` is called a *type tag* and can be used to refer to the constructed domain.

We use `typedef` here to map three type names onto the domain `pt`: `polar_point`, `xy_point`, and `twisted_point`. We now have four names by which we can refer to the same domain, and they are used, below, to declare a group of type-compatible variables: `pp, xp, tp`, and `sp`.

```
typedef struct pt {float c1, c2;} polar_point, xy_point;
typedef pt twisted_point;

polar_point pp;
xy_point xp;
twisted_point tp;
struct pt sp;
```

Exhibit 15.18. Merged domains in Pascal.

We declare two new type names, (`LengthInFeet` and `LengthInMeters`) by mapping them onto the domain `Real`. These two types are synonyms for each other and for `Real`.

```
TYPE  LengthInFeet = Real;
      LengthInMeters = Real;
```

Pascal was developed at about the same time as C. Although Pascal does support several type constructors ("`array`", "`record`", "↑", and enumeration), it does not permit the programmer to define a mapped domain with distinct semantics. A new type, D', can be mapped onto an old type, D, as shown in Exhibit 15.18. Objects belonging to D' are compatible with functions defined for D, and objects of type D are compatible with functions defined for D'. The two external domains are merged into one internal domain. Exhibit 15.19 shows examples of function calls on three Pascal types that belong to the same domain.

In this situation, Pascal does not help the programmer to make the semantic distinction between variables that represent reals and lengths, nor between different kinds of lengths. The type names are full synonyms in all contexts, and the semantics of the two external domains are merged. The fact that the type definitions in Exhibit 15.18 are not type constructors can be attributed to the lack of full understanding of domains and mapping in the early 1970s.

Exhibit 15.19. Function calls with merged domains in Pascal.

Three variables are declared, one of each type defined in Exhibit 15.18. A new function, `FeetToMeters`, is defined for lengths. It accepts a length, in feet, and converts it to the corresponding length, in meters.

```
VAR r:  Real;
    f:  LengthInFeet;
    m:  LengthInMeters;

FUNCTION FeetToMeters( f:LengthInFeet):LengthInMeters;
    BEGIN  FeetToMeters := f * 12 * 0.00254  END;
```

These variables are fully type-compatible; they belong to the same internal domain. Real functions, such as "*" and "/", and length functions, such as FeetToMeters, can legally be applied to any combination of lengths and reals. The following function calls and assignments are all legal, although those in the second row are semantically invalid.

```
r := FeetToMeters(f);      r := f;           f := 3.89;
f := FeetToMeters(m);      r := m * 5.0;     m := f / 2.0;
```

Exhibit 15.20. Creating a compatible type name by mapping.

The new type name `length` is declared as a synonym for `real` or `float`.

Ada	`subtype length is float;`
Pascal	`TYPE length = real;`
C	`typedef float length;`

The connection among domains, type names, and representations in Pascal and C is confusing. We may have one domain associated with more than one type name, and more than one domain associated with one structural description, as in Exhibit 15.14. The number of distinct domains is equal neither to the number of type names nor the number of different structures defined.

Mapped Types that Form Distinct Domains

The confusing connection among domains, type names, and representations was "cleaned up" in Ada. The type structure in Ada is richer than either Pascal or C. Ada permits the programmer to declare a type D', mapped onto D, but choose whether D and D' will be synonyms or name distinct domains. Ada provides two different declaration forms for using an old type to represent a new one.

The first form, marked by the keyword "`subtype`", creates a new name for part (or all) of the old domain [Exhibit 15.20]. The names become synonyms as they would in the corresponding Pascal [Exhibit 15.18] and C [Exhibit 15.17] declarations.

The second Ada form, marked by the keyword "`new`", creates a *derived type* [Exhibit 15.21]. A derived type is a new domain that is at least partially distinct from the old domain.

An issue arises with using a new domain that was created by mapping. If the new domain were completely incompatible with existing domains, no functions would be defined on it, and you couldn't define any! We must use existing function definitions to define the set of operations appropriate for the new domain.

The primitive operations of subscript, part selection, and dereferencing are defined for all domains that are constructed using the type constructors "`array`", "`record`", or "↑". Given an object, O, of a new domain, we can use these primitives to "extract" parts of O that belong to old domains. Then we can use the functions defined for the old domains to implement the primitive functions for the new domain. We need a similar way to go from new new domain to old for mapped domains.

Exhibit 15.21. An Ada derived type is a new domain.

The type name `new_type` will represent a domain that is only partially compatible with the domain named `old_type`.

```
type new_type is new old_type;
```

Exhibit 15.22. Type casts in Ada.

The domain `tonnage` is derived from the domain `float`. Casts are predefined in Ada for derived types. Two casts are used here in order to permit mixed-domain arithmetic.

```
type tonnage is new float;

t1, t2:  tonnage;    -- Two variables of type tonnage,
ff :  float1;        -- and one of type float.

ff := 3.2;
t1 := tonnage( float(t2) * ff );
```

This could be in the form of a general casting operation, "`REP`", that would take an object of a new mapped domain and "extract" the object that represents it in the old domain. `REP` would be a compile-time operation, changing only the domain label and not the representation. An inverse cast, "`MAKE`", would also be needed to relabel values computed in the old domain as elements of the new domain. `REP` and `MAKE` are analogous to the Pascal functions `ord` and `chr`, but they describe the general relationship between a mapped domain and its representation, rather than the specific relationship between characters and integers [Exhibit 15.22].

Ada supports type casts between representing and represented types. A cast is called by putting the target domain name in front of the value to be relabeled. `REP` and `MAKE` casts are both written this way.

Derived types could have been defined in Ada as completely independent new domains, but they were not. One-way compatibility was retained. We summarize the compatibility rules here, for a new type T derived from an old type R:

- Literals of type T are written exactly like R literals.
- Functions predefined for type R can be applied to type T.
- Functions defined for type T cannot be applied to type R.
- A value of type R can be explicitly cast to type T, and vice versa.
- Such a cast *must* be done before values of types R and T can be mixed in an expression or an assignment statement.

These rules implement one-way semantic protection. Objects of `new_type` can use functions for `old_type`, but not vice versa [Exhibit 15.23]. The advantage of this partial compatibility is that the basic definitions for a new ADT can be a bit shorter, because explicit casts are not required. Literal values, also, don't need to be cast to the new type. The costs of this compatibility are that the Ada programmer must learn a complex set of type compatibility rules, and that the compiler is

Exhibit 15.23. Domain compatibility rules in Ada.

Two new types are defined to be represented by floats. Length is a synonym for float and distance is semantically distinct. Variables are defined of the three types.

```
subtype length is float;
type distance is new float;
flo:  float;
len:  length;
dis:  distance;
```

The following expressions are all legal (put is an output procedure predefined for floats).

```
flo := 8.1;          len := 8.1;          dis := 8.1;
flo := flo+2.0;      len := len+2.0;      dis := dis+2.0;
put(flo);            put(len);            put(dis);
flo := flo * flo;    len := len * len;    dis := dis * dis;
                     len := flo;
                     len := len + flo;
```

The following statements have type errors.

`dis := flo;`	An explicit type cast, as in: `dis := distance(flo);` must be done before the assignment.
`dis := dis + flo;`	Mixed type arithmetic is not allowed. One operand must be cast before the operation.

unable to detect half of the unintentional type errors. Further, after the basic functions for a new domain have been defined, those functions can and should be used *exclusively* in defining further operations. Compatibility with the old type is no longer needed or desirable, but it is still permitted by Ada. This seems to be a real defect in Ada's type system.

15.5 Type Casts, Conversions, and Coercions

In the previous section we mentioned the topic of type casts briefly [Exhibit 15.23]. Here we explore the nature of casts and also examine other kinds of conversion processes. We examine what happens to the physical type and the semantic properties of the converted object during the conversion. In Section 15.6, the type compatibility and conversion rules in a number of common languages are described in detail.

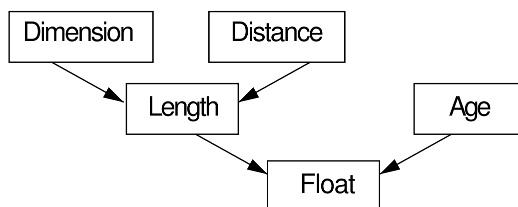
Conversion processes can be classified into two categories: conversions and casts. Processes in both categories change some property of their parameter. Both might be called for explicitly or invoked automatically. The words "cast", "conversion", and "coercion" are all in common use, but

Exhibit 15.24. A hierarchy of mapped domains.

The following set of domains are all represented internally as floats, but they have different operations defined.

- Dimensions (but not distances or ages) can be multiplied to compute areas and volumes.
- Distances (but not ages) can be added to compute a total distance.
- Neither dimensions nor distances can be added to floats.
- Ages can be subtracted from each other to yield a float (age difference) and added to floats to produce another age.

The domains "length" and "age" were defined by mapping directly onto "float". Domains "dimension" and "distance" were defined by mapping onto "length". We have thus created this tree of domains:



have fuzzy meanings.[9] We use all three terms, and give them distinct meanings. We define the terms briefly here and explain them at length in the following sections. A *type cast* is a change in semantic labeling involving one domain that is mapped onto another. A *conversion* is a change in the size, encoding, or reference level of an argument. A *coercion* is a conversion or a cast that is invoked automatically by the translator.

15.5.1 Type Casts.

Section 15.3.3 dealt with the domain relationship called "domain mapping", in which a new domain is implemented by setting up a correspondence between its members and members of some existing domain. Typically, the old and new domains are semantically unrelated. Several domains, D'_1, D'_2, etc. may all be mapped onto one representing domain, D. Mapped domains can thus form a tree-structured hierarchy [Exhibit 15.24].

A *type cast* is a "conversion" between mapped domains. It is a curious thing: it leaves the bits of a value unchanged but alters its domain label, thereby changing the semantics of the object. Type casts convert values in a represented domain, D', to or from values in the representing domain, D.

[9]The word "cast" is used in books about C to include all explicitly written casts and conversions.

Exhibit 15.25. Pascal casts for integer-mapped domains.

Represented Domain D'	Representing Domain D	Casting Functions D' to D	D to D'
enumerated type	integers	ord	Not primitive
character	integers	ord	chr
addresses	integers	Prohibited	Prohibited
truth value	integers	ord	Not primitive

For example, binary integers are used to represent characters in Pascal. The function `chr`, which takes an integer and returns the corresponding character, is a cast; so is its inverse, `ord` [Exhibit 15.25].

The operation of casting happens entirely at compile time. A cast actually does nothing to its argument except relabel the argument value with a different type-object. The purpose of a cast is to communicate to the compiler that it is meaningful to use an object in what would appear to be the wrong domain context. This prevents the compiler from generating a type-error comment. The compiler does not generate run-time code for a cast operation—the physical representation of the cast argument is already appropriate for the target domain and does not need to be changed.

Once the basic functions and data structures of an ADT are defined, an application program works within the defined domains and rarely needs casts. However, the ability to cast a value between the represented domain and the representing domain can be very important in defining the basic ADT functions. For example, with the mapped domains "imaginary" and "real", one needs to use "real" operations in order to define "imaginary" arithmetic. To do this one needs to cast the "imaginary" operands to "real", do "real" arithmetic, then (in some cases) cast the result back to "imaginary" [Exhibit 15.26].

Exhibit 15.26. Using casts in a mapped domain.

Using Ada, we define imaginary numbers by mapping them onto the reals. (The real domain is called "`float`" in Ada. Addition and multiplication of imaginary numbers is then defined in terms of arithmetic on the reals. Explicit casts are used here to clarify the semantics, even where they could be omitted.

```
type imag is new float;

function "+" (q,r:  imag) return imag is
    begin return imag( float(q) + float(r) ) end "+";

function "*" (q,r:  imag) return float is
    begin return -1 * float(q) * float(r) end "*";
```

Exhibit 15.27. Implementing a cast in Pascal.

If we accept the semantics that 0 represents `FALSE` and all nonzero integers represent `TRUE`, the following code converts an integer to a truth value:

```
FUNCTION IntToTv (k:  integer):  Boolean;
BEGIN if k = 0 then IntToTv := FALSE else IntToTv := TRUE END;
```

Pascal places limitations on casting which make the language clumsy or inappropriate for systems programming. Certain important type casts are not supported at all. In one such case, we can write a simple function that converts integers to truth values [Exhibit 15.27]. But the casts for address to integer and integer to address are prohibited altogether and cannot be implemented *within* the defined semantics of the language. Thus any kind of address arithmetic is prohibited in Pascal, causing some systems programmers to avoid the language.

We say that a cast is a *promotion* if it moves from domain D to domain D', like `chr`. A *demotion*, such as `ord`, casts a value from D' to D.

A *promotion cast* adds a layer of semantics to the object that it did not formerly have. With *demotion casts*, there is a loss of semantic information. The cast strips off the semantics of domain D', like an extra suit of clothes, exposing the underlying semantics of the domain D. If D was itself a represented domain, another demotion cast would strip off another layer of semantics.

Casts Are Essential. Both demotion and promotion casts are essential during the bootstrapping process that creates the functions for a new domain. Demotions permit operations defined for the old domain to be used on demoted members of the new domain. One must demote a value and operate on the underlying representation since there are, initially, no functions for the new domain. A promotion cast must be applied to the result of the computation to "lift" it back to the level of the new domain [Exhibit 15.28].

After this bootstrapping process is finished, elements of the new domain can and should be manipulated only by functions defined for the new domain. It would be desirable then to "seal off" the mapping relationship and prohibit further demotion and promotion casts.

Casters Beware! Casts are inherently dangerous operations; changing the meaning of an object should not be done lightly. It is legitimately done only in the process of implementing new domains, and in systems programming environments where the programmer is forced to deal with the underlying representations of objects in order to achieve acceptable efficiency.

Many languages use type coercion to change the type of an argument to the type expected by an operator. This is semantically meaningful if the two domains have related semantics, like integers and reals. But a type cast, by its nature, relabels a value with the identity of a domain that is usually unrelated, and thus changes the meaning of the value. A programmer writing an explicit cast presumably knows what he or she is doing and is taking responsibility for the meaning of the

Exhibit 15.28. Using casts in Ada.

Here we declare two mapped domains, `length` and `area`, and variables from the new and old domains:

```
type length is new integer;  -- Length and area form new domains.
type area is new integer;    -- Area and length are independent domains.
k:  integer := 15;           -- Declare and initialize integer k.
x:  length;                  -- x is a variable in the domain "length".
a:  area;                    -- a is a variable in the domain "area".
```

A type cast is called for by writing the name of the target type followed by parentheses enclosing an element from the original type. We may create a length-value out of an integer by using a promotion cast:

```
x := length(i)
```

A demotion cast must be used to define basic operations for lengths. Here we define an additional method for the primitive operator "*".

```
function "*"( x, y:  length) return area is        -- Note a
begin
    return area(integer(x) * integer(y));          -- Notes b, c
end "*";
```

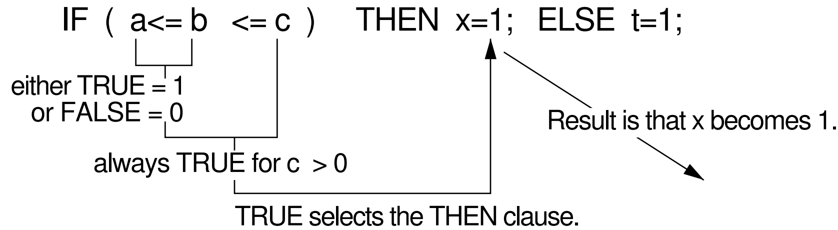
a. The function returns a value of type `area`.

b. The demotion cast, is used twice, `integer(x)` and `integer(y)`, so that the meaning of length multiplication may be defined in terms of multiplication on the underlying domain, `integer`.

c. The promotion cast, `area(z)`, is used to promote the underlying integer representation to the appropriate mapped domain. Note that `length` is not the appropriate domain.

Exhibit 15.29. Mangled meaning in PL/1.

Let us examine the automatic conversions triggered by the expression: a <= b <= c



1. Compare a to b; the answer is a truth value, represented as a bit.
2. Demote the domain of the truth value to the domain “bit string”.
3. Promote the domain of the bit string to the domain integer, to match the integer c and the definition of “<=”.
4. Promote the length of the integer to the length of c.
5. Compare the resulting integer to c. As long as c is positive, the result will always be `TRUE`.
6. Use this `TRUE` in the “if” test, selecting the “then” clause.
7. Store 1 in the variable x.

---

result. A flexible language must support explicit casts. However, a compiler has no understanding of the meaning of anything. It certainly cannot discern those contexts in which it is appropriate to change the meaning of an object.

A complication of promotion casts makes them particularly dangerous to invoke automatically. Many domains can be mapped onto a single representing domain. When a demotion cast is used on any of these mapped domains, it exposes the semantics of its single underlying domain. But a promotion cast might go in any of several directions. If a promotion cast is done by a compiler that cannot understand the code, it may utterly distort the meaning of a value. A combination of automatic demotion and promotion casts is likely to produce total nonsense, as in the PL/1 example in Exhibit 15.29

Consider what has happened here. The second item relates a mapped domain to its underlying representation and maps truth values onto length-1 bit strings; `'1'B` represents `TRUE` and `'0'B` represents `FALSE`. These domains do not have the same semantic intent. When a demotion cast is made, the semantic intent of the mapped domain is lost. In this example, when `FALSE` or `TRUE` is represented as a single bit and demoted to type “bit string”, it causes the original intent to be

"forgotten". It is no longer possible to tell that this bit string originally represented a truth value.

In the third and fourth steps, the single bit is promoted to integer and lengthened to the size of the integer `c` by padding it with leading zero bits. Lengthening is a semantically safe operation, but promotion is not. In the fifth step, the former truth value is used as an integer, even though this meaning is wholly inappropriate. The nonsense result of "a <= b <= c" is a result of the automatic invocation of semantics-changing casts, a demotion followed by a promotion.

15.5.2 Type Conversions

A *type conversion* changes one or more of the physical properties of the object: its size, its encoding, or its reference level. Examples of *encoding conversion* routines are the FORTRAN functions `INT` and `NINT`, which change a number from floating-point encoding to binary integer encoding, using truncation and rounding, respectively. Changing a short integer to a long integer, or vice versa, is a *size conversion*. A *reference conversion* substitutes a value where an address or pointer is given (by fetching) or an address where a value is given.

We are often concerned whether a conversion preserves all the information in a value, or whether some information is lost. We use the term *conservative* for conversions in which all of the information originally present is retained in the converted form, and *information-losing* for conversions that do not preserve all information.

Size Conversion

Size conversion changes the number of bytes used to represent a value without changing the value's encoding or its semantics. Some modern machines support integers of three or four lengths, and floats of two lengths. Many languages reflect this hardware structure by supporting types with the same encoding and different sizes. This produces a flexible language that can be used to achieve both time and space efficiency. However, when operands of different sizes are mixed in an operation, their sizes must generally be adjusted to match.

For example, C provides integers of at least three lengths, 1 byte, 2 bytes and 4 bytes. The 1-byte size, called "`char`", and the 2-byte size, called "`short int`" or "`short`", are needed to achieve acceptable storage efficiency. Moreover, short integers correspond to the hardware capabilities of personal computers, and so can be faster to fetch and faster in arithmetic operations than longs. Four-byte integers, called "`long int`" or simply "`long`", are needed to store many pointers and numbers greater than 32,767.

We call size conversions which lengthen the representation *promotions*, and those which shorten it *demotions*. Size promotions are always conservative; demotions lose information if the argument is large. Size adjustment is probably the safest and most useful kind of conversion, and the easiest to implement. Size adjustment is the only kind of automatic conversion that is generally supported in Ada.

Exhibit 15.30. Reference level in C.

We use C to define and initialize several objects that have integer encoding, long length, and various ref-levels.

```
#define  SAMPLE 50000L   /* "L" makes this a long integer, ref-level 0.  */
long int k     = SAMPLE, /* Ref-level 1:  a variable.  */
         *pk   = &k,     /* Ref-level 2:  a pointer.  */
         **ppk = &pk;    /* Ref-level 3:  a pointer to a pointer.  */
```

Encoding Conversion

When an external domain has multiple internal encodings, an *encoding conversion* maps one encoding of an object onto its nearest counterpart in the other encoding scheme. The ideal is that unconverted and converted forms are both representations of the same external object. This is more or less true for the conversions integer-to-floating point and floating point-to-integer that are predefined in many languages. If the machine implementation of floating point uses more bytes than implementation of integers in the same machine, the conversion from integer to floating point is conservative; all integers can be represented exactly in a long enough floating-point encoding.

In the other direction, floating point-to-integer conversion is defined to give as good a mapping as possible, although information is necessarily lost if the number has a fractional part or a large exponent. The answer returned from a conversion can be either the integer value nearest to the real value (conversion by rounding) or the first integer value closer to zero than the real value (conversion by truncation). Some languages (for example, FORTRAN and Pascal), provide primitives for both versions, others provide only one. For example, Ada) supports only conversion by rounding.

Reference Conversion

Let us define the *reference level*, or *ref-level* of an argument to be the number of times a dereference operation must be applied to arrive at a pure value. Thus the ref-level of "3" is 0, and the ref-level of a pointer pointing to a variable that contains this "3" is 2 [Exhibit 15.30]. A *reference conversion* takes an argument of one reference level and returns a result with a difference reference level (but the same encoding and size).

A reference demotion takes a storage object and returns a program object. A reference promotion takes a program object and returns a storage object. Demotions are used all the time; promotions are rare.

15.5.3 Type Coercion

Conversion processes can be invoked two ways: explicitly or automatically. A call on a conversion function or a type cast that is written in the source code is an *explicit conversion process*. A *coercion* or *automatic conversion* is one that is invoked by the compiler but does not appear in the

Exhibit 15.31. Coercions and explicit conversions in Pascal.

Line (a) contains two explicit type casts. The function `ord` casts its argument from the domain "char" to the underlying domain "integer".

The addition in line (b) will trigger a coercion from integer encoding to real (floating point) encoding. The compiler will invoke the encoding conversion because no machine instruction can add an integer to a real without conversion. Converting the value "makes sense" of the source code.

The right sides of lines (a) and (b) trigger reference coercion, to extract values from the variables `c`, `Number`, and `DigitValue`.

```
Var c:  char;
    Number:  real
    DigitValue:  integer;

DigitValue := ord(c)-ord('0');       { a }
Number := Number*10.0 + DigitValue;  { b }
```

These two lines could appear in an input conversion routine that manually converts numbers in an ASCII input stream to floating-point encoding. (This is not the best way to do the job, though.)

source code. Language definitions generally specify a set of possible coercions, which is a subset of the conversions that can be applied explicitly.

A translator will attempt to coerce an argument when the source code does not "make sense" as written, because the domain of some argument fails to match the domain of an operator or function applied to it. If one (or a series) of the permitted coercions will make the argument fit the context, it (they) will be invoked [Exhibit 15.31]. Coercions are used in many languages to change the encoding, the size, and/or the reference level of an argument.

When a coercion must be done because of mixed-type operands, a translator could theoretically choose to convert either argument to the type of the other. However, it is usually true that one direction is conservative and the other is not. A conservative conversion process is always preferred over the inverse information-losing process because it is unlikely to distort the meaning of the argument. This is particularly important when the translator (which does not and cannot understand the meaning of the code) is invoking the conversion. A programmer might find that the nonconservative conversion does exactly what is needed when it discards some information. But the translator has no way of knowing all of a programmer's intent, and it must always "play it safe" by choosing a conservative transformation when possible.

Size Coercion

When a language supports multiple sizes of types with the same encoding, keeping track of the size of objects and manually converting between sizes is burdensome and distracting. It becomes more burdensome as the number of related types grows. Such languages typically use size coercion

Exhibit 15.32. COBOL fixed-point size specifications.

PICTURE	Type Denoted
999	A three-digit integer.
9V99	A three-digit number with two decimal places.
999PP	A number with two implied trailing zeros, in the range from 100 to 99,900.
PP999	A number with a leading decimal point followed by two implied zeros, in the range from .00999 down to .00001

(promotion) in any context where the argument sizes are mixed. Size demotion is not a conservative operation, so arguments are only coerced to the smaller size when a long value must be stored in a small storage object.

For example, numbers may be declared in COBOL to be from one to many digits long, with a decimal point anywhere, either within these digits or any number of places before or after the string of digits. A "PICTURE" clause which conveys all this information is included in each variable declaration [Exhibit 15.32]. COBOL's type flexibility is actually important in many business applications where the number of decimal places maintained before rounding is vital. But in order to use two numbers of different sizes in an arithmetic expression, one number must generally be converted to the size of the other. COBOL does this automatically; it would be a distracting and awkward process for a programmer to manage.

Finally, a modern language must support the use and manipulation of character strings, either as part of the language or as part of a standard library. The string facility must conveniently provide for reading strings of varying lengths, storing them in variables, and for operating on strings of different lengths.

Storage can be handled in two ways. Dynamically typed languages (APL, SNOBOL) and BASIC store strings in a dynamically allocated string storage area and implement string variables by binding the variable name to a string value. This permits string length to vary as needed. Typed languages (C, Pascal, FORTRAN, Ada) preallocate storage for all variables and allocate fixed-size areas to hold strings. Strings stored in these areas are adjusted in length by truncation, marking the right end with a string termination symbol or padding the right end with blanks.

A language (or string library) should handle input, output, padding (lengthening), and truncation (shortening) operations automatically. Unhappily, many languages do not. Pascal, for example, forces programmers to write their own string input routines and manually pad or truncate strings to make them fit into the fixed-length spaces.

Encoding Coercion

Encoding coercions are important when a program deals with semantically related domains with different encodings. FORTRAN originally did not implement automatic conversions in arithmetic expressions. (That is, mixed type expressions were prohibited.) This prohibition was dropped in FORTRAN 77 because of popular demand. Experience showed that it was a terrible nuisance for

programmers to concern themselves with explicitly converting a 1 to a 1.0 for a computation or a 4.0 to a 4 in order to use it as a subscript.

APL and BASIC go farther. They don't even require a programmer to make a distinction between integer numbers and real numbers but choose an appropriate encoding dynamically and coerce numbers when necessary.

Integer-to-real and real-to-integer conversions are both applied automatically in many languages (FORTRAN, C) but under different circumstances. The conservative conversion is always used in cases where arithmetic is done on operands of mixed type. This implements the intuitive requirement that information, and therefore meaning, should be preserved during all operations in order that the ultimate result be meaningful and as accurate as possible.

The information-losing conversion is only applied automatically when absolutely necessary, that is, when the programmer orders that a floating-point number be stored in an integer location. If the programmer did this purposely, she or he is likely to be using this assignment command as an easy way to invoke the truncation function. If the programmer did not realize this was a mixed type assignment, the conversion is likely to result in a program error.

One might claim that such an information-losing conversion function should never be automatically invoked and that a programmer should explicitly invoke an information-losing conversion if one is needed. This is, in fact, the rule adopted by Pascal. Ada goes even further—it does not, in general, invoke encoding coercions of any kind.[10]

Reference Coercion

A reference coercion happens when a program object is supplied in a context that requires a storage object, or vice versa, and the translator "makes good" the discrepancy. Automatic dereferencing, invoked by language translators in most languages, is a downward coercion; an address is supplied in a context that requires a value [Exhibit 15.35]. The translator fetches the required value from the given address. In simple contexts, such as the arithmetic statements in Exhibit 15.31, reference coercion is a real convenience and eliminates the explicit dereferences that would otherwise clutter the code. However, in linked list programs, which contain many situations in which automatic and explicit dereferencing must be mixed, reference coercion leads to endless confusion. It requires great care to write the correct combination of explicit and implicit dereferences in complex situations. The last two lines of code in Exhibit 15.35 illustrate the nonintuitive nature of expressions that combine explicit dereference ("*") and reference coercion.

As seen in Chapter 6, languages exist that do not do automatic dereferencing. However, it is so omnipresent that programming students tend to accept it as the only normal convention and forget that a conversion process is happening.

The opposite coercion, from value to reference, is rare, since it amounts to creating a nameless variable and storing a value in it. Under restricted circumstances it is done automatically in FORTRAN. All function parameters in FORTRAN are passed by reference, but a programmer *is*

[10] Curiously, there is one exception to this in Ada; a integer value will be coerced to a real type if it is multiplied by a real value.

permitted to call a function with a pure value as the argument. In this situation FORTRAN must coerce the value to an address by storing it somewhere and passing the address to the function.

Although this is the extent of reference coercions in the familiar languages, other languages (EL1, Aleph) have been designed which apply a second reference coercion automatically when needed to fit the context. We are all familiar with the automatic dereferencing of variables to get values. These languages will dereference a pointer to get a value, if it fits the context.

15.6 Conversions and Casts in Common Languages

This section gives a description of the type transformations that are implemented in COBOL and in five languages from the ALGOL family (FORTRAN, C, PL/1, Pascal, Ada). In all except Ada, if a conversion is conservative it will be performed automatically when operands of mixed type are used in an expression. A nonconservative conversion will only be performed when the programmer directs that a value be stored in a variable of the lesser type. Ada has more restrictive conversion rules.

15.6.1 COBOL

Both size and encoding conversions are performed automatically whenever they are necessary to perform a specified operation. In practice this means almost constantly, for the following reasons: first, objects of many sizes are usually declared; second, the encoding ASCII-string is used for most numeric variables, but the encoding packed-decimal or binary-integer must be used for arithmetic. Thus the language processor is constantly involved in adjusting sizes and changing encodings. The conversions it uses are normally conservative. An information-losing size conversion will only be performed when the programmer directs that a value be stored in a variable too small to hold it, in which case the value will be truncated.

Because COBOL does use automatic size demotion, it is possible to lose the high-order digit(s) of a number by attempting to store it in a field that is too short. When this happens in an arithmetic statement, an error indicator is turned on. The programmer may (or may not) choose to test the overflow indicator in the program. In a simple assignment statement (`MOVE`), though, no indication of error is given. The programmer must be careful to avoid this, as the language does not provide a reasonable level of error checking.

15.6.2 FORTRAN

The standard language supports a variety of types. In addition to the usual `CHARACTER`, `LOGICAL`, `INTEGER`, and `REAL`, there are two numeric types, `DOUBLE` (double-precision floating point) and `COMPLEX`. No conversions or casts are defined for the type `LOGICAL`.

Two explicit casts are provided as standard functions in FORTRAN 77. These are never applied automatically.

`ICHAR(ch)`	Demotes a character to the underlying ASCII code.
`CHAR(i)`	Promotes an integer in the range 0..127 to a character.

Only two primitive types share both semantics and encoding; thus only one pair of size conversion functions is built into the language. The conversion marked "*" is conservative.

*	`DBLE(r)`	Promotes the size from real to double precision.
	`REAL(d)`	Demotes the size from double precision to real.

Encoding conversions are more numerous because three distinct numeric types are available, integer, floating point (real and double), and complex. The conversions marked "*" are usually conservative.

	`INT(r)`	Converts real, double, or complex to integer by truncation.
	`NINT(r)`	Converts a real or double to an integer by rounding.
*	`REAL(i)`	Converts an integer to single-precision floating point.
	`REAL(c)`	Converts complex to real by discarding the imaginary part.
*	`DBLE(i)`	Converts from integer to double-precision floating point.
*	`CMPLX(x)`	Converts from integer or real to complex.
	`CMPLX(d)`	Converts from double precision to complex.

15.6.3 Pascal

The standard language supports a few encodings that can occur in types of different sizes: character strings can be declared to be of any length, and character strings and Boolean arrays can be either packed or unpacked. String lengths are adjusted for comparison and assignment as necessary. Character strings and Boolean arrays are packed and unpacked automatically, when necessary, in a way that is quite transparent to the programmer.

Standard Pascal provides two types, integer and real, that implement domains with related semantics. The conversion functions accessible to the programmer are:

`round(r)` converts a real to an integer by rounding.

`trunc(r)` converts a real to an integer by truncation.

No function is provided to explicitly convert an integer to a real. Integer literals are also considered to be real literals, and the values of integer variables are converted automatically when necessary. Reals are never automatically converted to integers, because that is usually an information-losing conversion. This conversion must always be done explicitly.

Implementations of Pascal for microcomputers typically support two more types with integer encoding, a second size (so that both 2-byte and 4-byte integers are provided) and the type "unsigned integer" (frequently used to implement bit masks). Conversions are provided that promote short integers to longer ones. These are conservative operations and are applied automatically in some common implementations. Casts from unsigned to integer may also be performed automatically. When these are combined with the normally safe length changes, they can result in unintended

changes of meaning as is typical with automatically applied casts. It is in these areas that Turbo Pascal deviates most strongly from the standard language. Standard Pascal's semantic protection is lost in order to gain access to the underlying representation and convenience in handling mixtures of the three integer-encoded types.

15.6.4 PL/1

In order to understand Ada's restrictions on automatic type conversion, it helps to understand the effects of the generalized conversion rules in PL/1. Extensive, generalized, automatic conversion is supported, so that a PL/1 translator will convert any type to any type. The language designers perceived meaningful domain relationships between several pairs of primitive types (real—integer; bitstring—anything; numeric character string—number; bit—truth value; shorter object—longer object of same encoding). Conversions were defined for all of these relationships and are invoked automatically by the translator, singly or in series, whenever the programmer codes a mixed-type operation. Because the conversion from any type to any type is defined, expressions that are syntactically illegal in many other languages become legal in PL/1 and produce some surprising or nonsensical results [Exhibit 15.29].

Since any PL/1 object is type compatible (by conversion) with any other object, the language and translator cannot use domain checking to help the programmer achieve semantic validity. This is a great loss, and it far exceeds the value of automatic conversion mechanism. Coercions are convenient but not necessary, since explicit conversions could be used instead.

15.6.5 C

Basic Types in C. C implements a large variety of types semantically related to integers. Integers may come in three lengths, `short int` (2 bytes in the ANSI C standard), `long int` (4 bytes), and `char` (1 byte). The type name `int` refers to either `long` or `short`, whichever is more efficient for the hardware.[11] The type name "char" means "1-byte integer"; the semantics of chars are not differentiated from the semantics of integers. Integer operations can be applied to chars and vice versa. The external domains "character" and "1-byte integer" are thus merged internally in C.

An integer of any of the three lengths may have a sign or not; unsigned objects are created by including the type modifier `unsigned` in the declaration. Taken together then, there are six integer domains with eight type names.

The unsigned and signed types form distinguishable but not independent domains. In some cases different code will be generated for values of the two varieties. In particular, the methods for promoting the length of signed and unsigned integers are different (details are given below). On the other hand, the machine instruction for integer "+" will be used by C to translate "`x + y`" whether

[11] In K&R C, the lengths of short and long integers were not fixed. The only rule was that shorts could not be longer than integers and longs could not be shorter.

these numbers are signed or unsigned. This will normally carry out the programmer's intent.[12] A general rule for the safe use of these types is to use signed integers for numeric computation and unsigned for everything else (especially bit masks and addresses). Because unsigned types directly reflect the bit-string nature of computer storage, we considered them to be the basic domain and assert that signed values are mapped onto the unsigned. Thus we consider an int-to-unsigned cast to be a demotion, and unsigned-to-int a promotion.

Floating-point numbers of two lengths are implemented: float (usually 4 bytes) and double (usually 8 bytes).

Truth values exist and are mapped onto the integers. They have no separate type name but are created by comparison operators and used by conditionals, as in any language. `FALSE` is represented by "0", `TRUE` by any nonzero integer. A "1" is generated when the system must create a `TRUE` value. Truth values do not form a distinguishable internal domain, as their semantics are not differentiated from the semantics of the underlying type, `int`.

Casts in `C`. `C` does not distinguish between conversions and casts, although neither the processes nor their semantics are similar. Both are called "casts" in reference books, which are likely to explain that "a cast performs a type conversion". All conversions and casts may be explicitly invoked using the same syntax; the name of the target type is written in parentheses before the name or expression denoting the value that is to be changed [Exhibit 15.33].

Size Conversions in `C`. Size promotions may be done explicitly and are also done automatically when operands of mixed lengths are combined. Also, short integers (int and unsigned) are promoted to integer, and floats are promoted to double[13] when they are passed as parameters to a function. Within the function, parameters may be declared using the promoted or the original, nonpromoted, type. When the latter is done, the argument will be automatically demoted again. Characters are promoted and demoted at the convenience of the translator, whenever they are manipulated.

An unsigned value (integer or character) is promoted by padding it with 0 bits. A signed integer is promoted by sign-extension; that is, by padding the high-order end with copies of the high-order bit of the value. This retains the sign of the object and its absolute value if it is an integer. It is thus the semantically correct way to promote a signed integer. Signed characters may be promoted by either method, depending on the translator. Values are demoted by truncating the high-order end, an inherently risky operation.

Encoding Conversions in `C`. Two encoding conversions are implemented to convert floating-point numbers to integers and vice versa. These may be invoked automatically, or explicitly using the "casting" syntax in Exhibit 15.33. The conservative conversion, from integer to floating point,

[12] A nice aspect of two's complement encoding for negative numbers is that no special provision needs to be made for the sign during an addition operation.

[13] Non-ANSI only.

Exhibit 15.33. Syntax and semantics for casts and conversions in C.

```
int i;
unsigned u;
char c;
float f, g;
```

`(int)u`	Promote an unsigned to an int, no change in bit pattern. The result is undefined if the unsigned value is larger than the maximum int value. However, the operation will almost certainly be carried out by doing nothing! The high-order bit will simply be reinterpreted as a sign.
`(unsigned)i`	Demote an int to an unsigned, no change in bit pattern. The result is undefined for negative integers. However, the bits of a negative number will probably be unchanged, and the sign bit will be reinterpreted as a large positive power of 2.
`(int)c`	A size promotion from one byte to int (2 or 4 bytes).
`(float)i`	Convert from integer encoding to a floating point.
`(int)f`	Convert encoding from float to integer by truncation.
`(int)(f+g)`	Convert the sum from float to integer by truncation.
`i = (int)f+g`	Convert the float `f` to an integer by truncation, then convert it back to a float in order to add it to `g`. The result is a float; convert it to an int by truncation and store it in `i`.

is applied automatically when mixed type operands are used in an expression. This conversion might be automatically applied in combination with promoting the length of one of the operands.

Coercion in C. One of the anomalies of C syntax can be best understood in terms of reference coercion. A pointer may be set to point at a variable by assigning the address of the variable to the pointer. One indicates that the address, not the value, of the variable is to be used by writing an "`&`" before the variable name. This inhibits the reference coercion that normally would have been applied to the variable. Pointers may also point to arrays and functions, but in these cases no reference coercion ever happens, and the "`&`" must not be written [Exhibit 15.34].

Pointers, explicit dereferencing, reference coercion, and inhibited coercion are all combined in Exhibit 15.35. Arithmetic operators in C operate on numeric values, not on addresses. A commonly useful complex data structure involves a pointer to an array of pointers which in turn index an array of data objects. Here we declare such a structure and show some code that uses the structure. Note that in C, adding two values is a legal operation but adding two addresses is not. Adding an integer to a pointer which represents an array index is also legal.

Exhibit 15.34. Coercion complications in C.

Here we declare a variety of variables and pointers. The C syntax for setting a pointer to point at an object is shown on the fifth line. The "&" operator means "address of". Note that we do not use "&" on the last two lines.

`int k, *kk, *aa;`	Declare an integer and two pointers to integers.
`int a[5];`	An array of 5 integers.
`int f();`	This declares a function f (to be defined elsewhere) that returns an integer.
`int *ff();`	A pointer to a function such as f.
`kk = &k;`	Pointer kk is set to point at k, an integer variable.
`aa = &a[0];`	Pointer aa points at the first element of a, an integer.
`aa = a;`	This also makes aa point at the beginning of array a, because the name of an array is coerced to mean its first element.
`ff = f;`	Set ff to point at the beginning of function f.

Comments on the right in Exhibit 15.35 document the dereferences; each assignment statement is echoed, with code letters replacing the variables. The code "*E*" marks each explicit dereference, "*I*" marks each inhibited dereference, "*N*" marks contexts in which no dereference takes place, and "*C*" marks each variable where C applies reference coercion. Where multiple dereferences happen, a list of code letters is used; these should be read left-to-right.

Note that most ordinary expressions are marked by "*C*". For example, in the expression `(index1 < index2)`, both pointer operands represent storage objects and both are coerced to pointer values which are then compared.

The C language does not provide for handling array values coherently, nor for handling pure values of function types. When the programmer writes an assignment involving an array or a function, the translator makes sense of the request by inhibiting the automatic coercion (dereference) that normally would have taken place on the right side of an assignment statement. This is a double negative situation: automatic inhibition of an automatic coercion results in no action at all. The result is that the address originally given is assigned to the pointer.

The expression on the last line of Exhibit 15.35 is daunting; its meaning can best be ascertained by making careful diagrams. This illustrates the complexity inherent in the C approach to references and coercion!

15.6.6 Ada Types and Treatment of Coercion

Basic Types in the Standard Type Definition Package. As with C, a profusion of type definitions is included in the standard Ada package. These implement four external domains: truth value, character, integer, and real. The numeric domains are both implemented by several primitive types.

Exhibit 15.35. Explicit dereference and reference coercion in C.

```
int j, k, m;                    /* Integers.  */
int value_array[100];           /* An array of integers.  */
int *scanner, *temp;            /* Pointers to integers.  */
int *index_array[10];           /* An array of pointers to integers.  */
int **index1, **index2;         /* Pointers to pointers to integers.  */

/* Initialize three pointers.  */
scanner=value_array;            /* N = N */
index1=index_array,             /* N = N */
index2=&index_array[10];        /* N = I */
/* Make elements of the index array point at every tenth value.  */
while (index1 < index2)         /* C < C */
/* Compare the values of the pointers, not the values they point at!  */
{   index1++; scanner+=10;      /* N = C + 1 ; N = C + 10 */
    *index1 = scanner;          /* E = C */
}

/* Input two integers between 0 and 9; set pointers to those slots.  */
scanf("%d%d", &j, &k);          /* I, I */
index1=&index_array[j];         /* Set a pointer to jth index.  */
index2=&index_array[k];         /* N = I */

/* Swap two pointers in index_array.  */
temp=*index1;                   /* N = E,C */
*index1=*index2;                /* E = E,C */
*index2=temp;                   /* E = C /*

/* Add an int value to 3 times another int.  */
m= **index1 + *((*index2)+j)*3;  /* N = E, E, C + E((E, C) + C) * N */
```

Exhibit 15.36. Ada **integer type specification.**

Here are two type declarations that will map two new domains onto some primitive integer type with an appropriate length.

```
type line_count   is range 0..66;
type fathom       is range -5_000 ..  0;
```

If we declare an object of one of these constrained types, we can be sure of the range of values that can ever be stored it it. The variable `line_no` can hold only values between 0 and 66.

```
line_no:  line_count;
```

In contrast, these declarations for objects of the primitive integer types define variables whose range of values can vary from one implementation to another.

```
count:       integer;
population:  long_integer;
index:       short_integer;
```

All implementations support type `integer`, whose length, as in C, may vary from implementation to implementation. Types `natural` (like unsigned, range 0..`max_representable`), `positive` (like natural but excludes 0), `long_integer`, and `short_integer` might also be implemented.

If the programmer simply specifies the range of values that she or he intends to store in a variable, the translator will choose an appropriate size for the variable. Note the similarity to COBOL. The practical reason for using this facility is illustrated by Exhibit 15.36. The range of values that can be stored in the variable `line_no` is fixed and does not depend on the implementation. In contrast, the ranges of the other three variables depend on the translator. Using the type declaration with a range clause, therefore, produces more portable code and is considered better programming style. An attempt to store an out-of-range number, such as 85, into `line_no` would cause the run-time error "`CONSTRAINT_ERROR`" (called an "exception" in Ada).

As with the integer types, reals are implemented by a group of types whose lengths depend on the implementation. These come in two varieties, *floating-point types* (like reals in Pascal and floats in C) and *fixed-point types* (as in COBOL). The primitive type names are `float`, `long_float`, and `short_float` [Exhibit 15.37].

Truth values are supported: the Ada type `Boolean` is defined as the enumerated type `(FALSE, TRUE)`. No physical or semantic relationship exists between `Boolean` and any other type and, therefore, no conversions or casts are defined for Booleans.

Characters, specifically the ASCII characters, are the other predefined enumerated type. The mapping between the ASCII character sequence and the integers 0..127 is defined, but no functions are defined to cast characters to integers or vice versa.

Exhibit 15.37. Ada **real type specification.**

Use of the keyword `digits` indicates that the new type is to be a floating-point type; `delta` indicates fixed point. The new type will be mapped onto a primitive real type with an appropriate length and encoding. The precision of a 4-byte floating-point value is about 7 digits; 8-byte floats provide about 16 digits of precision.

Floating-Point Declarations.

```
type mass     is digits 15;                          -- Needs double length.
type pressure is digits 7   range 0.0 ..  25.0;  -- Single length ok.
```

For fixed-point numbers, the declaration completely determines the minimum size of the representation, as does a COBOL `PICTURE` clause. The `delta` clause specifies the required number of decimal places of accuracy, and the `range` clause shows the number of places needed to the left of the decimal.

Fixed-Point Declarations:

```
type dollars  is delta 0.01  range 0.0 ..  10_000.0;
type voltage  is delta 0.1   range -12.0 ..  24.0;
```

The variable `dollars` can take on values between 0 and 10,000. These values will be binary approximations to these values 0.01, 0.02, etc. Successive elements in the approximation must not differ by more than .01.

Ada **Conversions and Casts** Ada is the only language we have considered here whose designers made careful distinctions among the various ways an object of one type may be "changed" into an object of another type.

Ada provides type casts that move between a programmer-defined mapped type and its underlying type. These are for the purpose of defining the semantics of the mapped domain and are never applied automatically. No casts are defined that move between predefined mapped types and their representations.

Explicit conversions are not needed between different size objects of the same encoding. The "safe" conservative size promotions are used freely and automatically, without any involvement on the part of the programmer. Indeed, when using the recommended programming style the programmer may not even be aware which size of a primitive type is used to implement his or her objects. Ada will promote the size of the smaller object automatically when objects of different sizes are mixed in an expression.

Representation conversion functions are defined for all numeric types to convert among the three basic numeric encodings, integer, fixed point, and floating point. A conversion is invoked by writing the name of the target type like a function name, and writing the value to be converted as the argument of the function. Example: integer(123.9).

Exhibit 15.38. Mixed type addition in Ada.

We extend the operator "+" to operate on one integer and one float by supplying an additional computation method for "+".

```
function "+" (x:  integer; y:float) return float is
begin
    return (float(x) + y)
end "+";
```

These representation conversions are available for explicit programmer use, but they are never used for automatic coercion. This greatly simplifies the problem of maintaining protected semantic domains. If conversions and casts are not applied automatically, the programmer's semantic intent can not accidentally be violated. The cost of this simplicity is inconvenience to the programmer, since mixed-type arithmetic is, therefore, not predefined. The Ada programmer must either use explicit type casts to do mixed-type arithmetic, or explicitly define each operator for each desired combination of mismatched argument types [Exhibit 15.38]. At best, this is a nuisance; at worst, it causes time and space inefficiency to translate and store the code for these simple and repetitive function bodies.

Non-Extensible Domain Relationships

We have seen that many languages have predefined domain relationships and will use these relationships when they invoke automatic conversions. PL/1 is the most permissive of the group, as it will automatically convert anything to anything. Ada is the most discerning of the group, and it will invoke only casts and length conversions automatically.

None of these languages permit programmers to limit or control application of a built-in relationship, and none permit them to define domain relationships of their own that will be automatically invoked. For example, a programmer could define a new representation for numbers and write a type conversion routine to convert integers to the new representation. But none of these translators will use the conversion automatically, the way the integer-to-real conversion is used.

Chapter 17 explores more modern and sophisticated type systems that permit the programmer to define more kinds of domain relationships.

15.7 Evading the Type Matching Rules

Type checking is an immense aid to the programmer. The more strict the type rules are, the fewer stupid mistakes and oversights will go undetected. The cost of this protection, though, is inflexibility. Occasionally a programmer needs to perform an operation that breaks the rules.

Applications that can use such flexibility include conversion of integer to floating point and efficient computation of a hash index.

Users do not normally write integer-to-floating-point conversion routines because compilers normally supply them as primitive functions. A systems programmer, though, needs a language that will permit him or her to deal with hardware-dependent number representations. The bits of an integer value must be tested, shifted, and masked during the conversion process. The value starts out as an integer, ends up as a float, and is nothing recognizable in between. A systems programming language needs the flexibility to refer to this object as both an integer and a float. Let us say that the number being converted must have a *dual type*.

Another example of the use of dual-typed objects is hashing. The intent of a hash function is to generate a random-looking but repeatable integer within a specified range. Think of the argument to the hash function as a bit pattern. Hashing should scramble these bits so that the various inputs generate uniformly distributed integers as outputs. An easy way to do this is to apply an operation that is normally meaningless for the data and take the resulting scrambled bits. For example, adding the right and left halves of a character string using integer addition would randomize its bits in a repeatable, efficient, and possibly useful way.

Languages that are ST#3 (all objects are typed, types do not overlap, and all function calls are type checked) do not let a program do such operations in a straightforward way. Many, however, provide declaration forms that function as an "escape hatch" by which the programmer can get around the normal typing restrictions. By using such an escape hatch, a programmer declares that a normally meaningless operation is meaningful in a program. These declarations bind multiple names (with different types) to a single storage object or to part of an object. This gives the programmer a way to circumvent restrictions imposed by the type checking system of the translator.

When doing this kind of operation, it is up to the programmer to make sure that the result is semantically valid. Further, any time a program depends on a particular underlying representation, and exploits that representation in a computation, it becomes nonportable code. Different compilers and different machines use different representations. An obvious example is the order of two bytes in the representation of a short integer. On an IBM PC, they will be arranged low-order, high-order. But on a Macintosh, they will be high-order, low-order. Any code that depends on the order of these bytes is nonportable.

Examples are given here in several languages of declarations that can bind dual types to a single object.

FORTRAN. The `EQUIVALENCE` declaration is an entirely unrestricted way to map one object onto another. No restrictions are placed on the relative sizes of these objects, or on the relative positions of their beginning bytes. `EQUIVALENCE` is used in a very restricted way in Exhibit 15.39 to associate two types with one variable. Lines (a) and (b) of Exhibit 15.39 declare four names: a real, an integer, a one-dimensional integer array, and a two-dimensional integer array. In line (c) we use `EQUIVALENCE` to map the real variable `Z` and the integer variable `INTZ` onto the same storage location.

Exhibit 15.39. `EQUIVALENCE` **in FORTRAN.**

`EQUIVALENCE` may be used to bind an additional name and type to any variable or any part of a subscripted variable. (The letters on the right relate each declaration to its explanation.)

```
REAL Z                              (a)
INTEGER INTZ, M1(100), M2(4,25)     (b)
EQUIVALENCE (INTZ, Z)               (c)
EQUIVALENCE (M1(1), M2(1,1))        (d)
```

The following pairs of lines refer to the same storage locations:

```
X = Z          Copy the dual-type value into a real variable.
K = INTZ - 1   Interpret the dual-type value as an integer and subtract 1.

K = M1(29)     Copy an array item into K.
K = M2(1,8)    Does the same thing as the line above.
```

Note that the subscript in the last line is (1,8) rather than (2,4) because FORTRAN arrays are stored in column-major order, not row-major order like most other languages.

Line (d), similarly, maps the two arrays onto the same space. These two arrays have a different number of dimensions but an equal total number of elements. We now have two names for the same array, allowing us to refer to it using either linear or two-dimensional subscripts.

COBOL. The `REDEFINES` declaration is like `EQUIVALENCE` except that the original object and the redefined type are required to be the same size. The redefinition must immediately follow the original declaration.

Pascal. In FORTRAN, we declare two variables, then say, as an afterthought, that they are one and the same object. This serves the purpose of attaching two types to an object but certainly does not implement the abstraction "dual type". In Pascal, we can explicitly declare a type with two meanings, then use it to declare variables.

This is done with a nondiscriminated variant record. The common portion of the variant record and the tag field are omitted altogether, producing a record that consists entirely of two or more variant parts. Outside of the context of Pascal, this kind of dual type is called a *free union type*. It is impossible to tell, at run time, what the semantics of a non-discriminated variant object is supposed to be. A program may use an object as first one type, then another, without using any conversions or casts.

The examples used above for FORTRAN `EQUIVALENCE` are rewritten in Exhibit 15.40 in Pascal as nondiscriminated variant records. Analogous type declarations are given for an integer-real variable (line b) and an array that can be accessed using either one or two subscripts (line c). The Pascal variant record syntax requires declaration of an enumerated type (line a) with the correct number of variants.

Exhibit 15.40. Variant records in Pascal.

```
TYPE TwoVariants = 1..2;                               {a}
     IntReal = RECORD CASE TwoVariants OF              {b}
                     1:  (IntName:  integer);
                     2:  (RealName:  real );
                 END;
     OneDim =    ARRAY [1..100] OF integer;
     TwoDim =    ARRAY [1..4, 1..25] OF integer;
     DualDim =   RECORD CASE TwoVariants OF            {c}
                     1:  ( vector:  OneDim );
                     2:  ( matrix:  TwoDim );
                 END;
```

Exhibit 15.41 shows examples of the use of these records to implement dual-type objects. We declare an `IntReal` variable, `R`, and a `DualDim` array, `A`. When `R` is used in a context that requires a real number, we refer to "`R.RealName`". In an integer context we write "`R.IntName`". Similarly, `A.vector[3]` and `A.matrix[1,3]` refer to the same location.

C Has Two Escape Routes. In C, the `union` type constructor builds a free union type, like the nondiscriminated variant record in Pascal. The syntax for the `union` is similar to but simpler than the Pascal syntax for variant records. Exhibit 15.42 shows how the `union` type constructor can be used in C to accomplish the same goal as the FORTRAN code in Exhibit 15.39 and the Pascal code in Exhibit 15.41. Type declarations are given (line 1) for an integer-real variable and (line 4) for an array that can be accessed using either one or two subscripts.

Exhibit 15.41. Use of a Pascal nondiscriminated variant record.

```
VAR R: IntReal;
    A: DualDim;
    X: Real;
    K: Integer;

X = R.RealName      Copy the dual-type value into a real variable.
K = R.IntName - 1   Reinterpret dual-type value as integer; subtract 1.

K = A.vector[29]    Copy the second array item into K.
K = A.matrix[2,4]   Does the same thing as the line above.
```

Exhibit 15.42. Union data type in C.

```
typedef union {  long int_name;
                 float real_name;} int_real;        /* 1 */
int_real r;                                          /* 2 */

typedef int one_dim [100];
typedef int two_dim [25][4];
typedef union {  one_dim vector;
                 two_dim matrix;} dual_dim ;         /* 3 */
dual_dim a;                                          /* 4 */
```

- In line 1, type `int_real` is defined to be either a long integer or a floating-point number. Enough storage will be allocated to store whichever variant is longer. (They are often the same length.)
- Line 2 creates an object, `r`, of type `int_real`. To access it as a long integer we use the name `r.int_name`. To access it as a float we use the name `r.real_name`.
- Line 3 defines a type named `dual_dim` that can be either a one- or two-dimensional array. In either case, it has 100 integer elements.
- Line 4 declares an object, `a` of type `dual_dim`. We may access this array with either one or two subscripts, thus: `a.vector[k]` or `a.matrix[m][n]`.

Interestingly, the `union` type constructor is not the only way, or the easiest way, to create a dual-type object in C. This same end can be achieved by making two pointers (an integer pointer and a float pointer) point at the same object [Exhibit 15.43]. When the program needs an int, it can access the object through the integer pointer. To get a float, it can use the float pointer.[14] This is demonstrated in Exhibit 15.43. Here we allocate one storage object, named `k`, and make two pointers, `plong` and `pfloat`, point at it. Both pointers are initialized to point at `k`.

Any pointer in C can be cast to any other pointer type. This causes no change except a relabeling of the domain of the pointer. Specifically, a pointer cast does not convert the representation of the object to which the pointer points.[15]

Ada Closes the Loopholes. The designers of Ada put a high value on designing a semantically "safe" language, that would be wholly and completely ST#3. Free union types and unrestricted

[14]This "trick" is the basis of the object-oriented function dispatching in C++.

[15]The explicit cast on the initialization is not necessary at all in many older C implementations, and it is used only to avoid a warning message in newer compilers. The code would compile, run, and produce the same answer without the cast.

Exhibit 15.43. Using pointers in C to attach two types to an object.

In C, long integers and floats are usually the same length, so we use these two types to create a dual-type object, k. Then we compute a hash function by referring to k first as a float, and then as an int. We store a float value into k using pflot, and take it out using plong. This does not cause a representation conversion. Applying integer division to the misinterpreted value will not cause a representation conversion either. The result is scrambled bits; it is complete nonsense to divide a float value by an integer. Thus we have "hashed" the original number.

```
#define TABLESIZE 1000
long int j, k;
long int *plong=&k;
float *pflot = (float*)(&k);

*pflot = 3.1;              /* Put a floating-point value into k.  */
j = *plong % TABLESIZE;    /* Apply mod to k to calculate a value between
                                zero and the size of the hash table.  */
```

The programmer who uses this coding trick must do so carefully; any code that casts the type of a pointer, or depends on the representations of two types to be the same length, is nonportable.

casts are ways to evade the constraints imposed by strong typing, and they are not permitted in Ada.

Ada does support type casts, specifically, those necessary casts between types that are related by mapping. Type conversions are also supported between pairs of numeric types such as integer and real. These conversions must be invoked explicitly, using the same syntax as for type casts. But, unlike C, Ada does not support unrestricted casts from any type to any type. Thus the pointer trick will not work in Ada.

This surely increases both the readability and the portability of Ada programs. Type changes must be explicitly stated, eliminating confusion about intent. By eliminating free union types, access to the actual bit-level implementation of Ada objects is shut off, forcing programmers to write implementation-independent code. The cost is that flexibility to do some useful things is lost. The advantage is more reliable code.

Exercises

1. What is a domain?

2. How were implicit domains represented in early computer languages?

3. What is a predefined domain? Which ones were supported by the original FORTRAN? What determined the domain of each variable?

4. How did ALGOL enrich the domain structure of previous languages?

5. Why is "typeless" sometimes used as a synonym for "dynamically typed" language?

6. When is the domain of a variable tested in a dynamically typed language? Why is it tested? Why is it tested at that time?

7. Can function applicability be controlled in dynamically typed languages? Explain.

8. Why can early C be considered a weakly typed language?

9. Why were domains given increasingly important roles as programming languages developed?

10. What is a subrange declaration? Why is it useful?

11. How has Ada enhanced the use of domains beyond Pascal?

12. What is type checking?

13. Briefly define strong typing, and explain how it can be, at the same time, a boon and a burden.

14. How has the definition of "strongly typed language" evolved?

15. What is an ADT? Why can't we define an ADT in C or Pascal?

16. What is an external domain? Internal domain?

17. What are internally merged domains?

18. What are the problems associated with internally merged domains?

19. When a new domain has been mapped onto an old one, is the semantic intent of the two domains identical? Why or why not?

20. Name two domains whose semantics are totally different, even though they are both sometimes represented as integers. Give an example of two values that might be represented by the same integer.

21. Why is Pascal considered a strongly typed language? Explain.

22. What is a "type constructor"? Give an example. How does a type constructor permit the programmer to define new domains?

23. Why are some domains which have the same structural description considered incompatible in Pascal?

24. What is the role of a `typedef` declaration in C?

25. How does Ada improve upon the type structure found in C and Pascal?

26. What is a type cast? Conversion? Coercion? Give an example of each in a language that is familiar to you.

27. For each of the following kinds of type "conversions", say whether the original and converted objects represent the same thing and whether the representation (bit pattern) is changed.

    a. integer to real
    b. The result of an addition used as a truth value
    c. long integer to short integer

28. Give and explain an example of automatic type conversion destroying the semantics of an object. The example may be drawn from any language that uses automatic conversion, but please say what language you are using.

29. What is an encoding conversion? Size conversion? Reference conversion?

30. Name two domains that contain some elements that have the same meaning. As an example, show a pair of values that correspond.

31. Explain the following statement: A conversion involves a run-time computation, but a cast happens entirely at compile time.

32. A data value in one domain is changed to a corresponding value with the same (or similar) meaning in a second domain. Is this a conversion or a cast?

33. Consider the Pascal statements given below. What semantic action is implicit in this fragment?

```
jj:  integer;
ff:  real;
begin
    ff := 3.0
    jj := 1;
    ff := ff + jj;
end
```

34. Would the following code in Ada (analogous to the Pascal code in question 33) be legal? Why or why not?

```
rr:  float := 3.0;
jj:  integer := 1;
begin
    rr := rr + jj;
end
```

35. What is the difference between a promotion and demotion cast?

36. Why are casts considered dangerous operations?

37. Consider these equivalent expressions, where b is an integer:

```
In APL: 8 > b > 3
In C or Pascal:  3 < b < 8
```

    The Pascal expression causes a compile-time error. What is it?

38. The APL and C expressions in question 37 will compile and run without errors. Evaluate the expression in either language using the value `b = 2`. What semantic distortion is committed by the translator that causes the result to be different from the programmer's apparent semantic intent? Why does it not produce a type error in these languages?

39. APL objects retain their type tags on the stack, and data types are checked by all primitive operators (unlike FORTH). Nonetheless, because statement labels are mapped onto integers, APL will sometimes execute code that is semantic nonsense and would be identified as a type error in other languages. Give an example of code that is legal in APL but that computes nonsense. Explain briefly what semantic error is being made. (Note: The fact that identifiers do not have types is not a semantic error.)

40. Explain what the semantic problem can be when a language applies two-way automatic conversion between a user-defined data type and the primitive data type used to represent it.

41. Explain or give an example of one way that C will permit a programmer to make a semantically absurd calculation.

42. Describe or give an example of a situation (other than that described in answer to questions 37 and 38) that would cause a compile-time error in Pascal but that would be legal at compile time in C.

43. Describe or give an example of a situation that would cause a run-time error comment in Pascal but that would not be considered to be an error in C.

44. Give an example of an error that Pascal cannot identify at compile time but will identify at run time. Why can't this error be identified at compile time?

45. What is meant by a conservative conversion?

46. When does a translator attempt to coerce an argument?

47. Why is size coercion essential in the use and efficiency of computer languages?

48. When are encoding coercions used?

49. In what sense is automatic dereferencing a downward coercion?

50. Why is it sometimes necessary to evade type compatibility rules?

51. Give an example of a "loophole" in a strongly typed language.

52. How have Ada designers created a semantically "safe" language?

53. Because FORTH objects lose their types when they are put on the stack, FORTH permits the programmer to violate the semantic intent of these objects, either by accident or on purpose. Explain and give an example of such a violation.

54. Happy Hacker is using a version of FORTH in which integers occupy 4 bytes. He has extended the semantics of the language to include a new type declarator, `REALVAR`, for real (floating-point encoding) variables, each taking 4 bytes. Happy wants code to compute the same function as in the Pascal code in question 33, and has written this:

    ```
    1 VARIABLE JJ
    3 REALVAR FF
    FF @ JJ @ + FF !
    ```

    Would this code work? If so, what answer would be left in `FF`? If not, why not?