1.1. Symbol descriptors

Symbol descriptors contain a reference to the identifier for which the symbol is created, a reference to the type descriptor and certain other information depending on the kind of the symbol.

Kinds of symbols include constant, type, variable, function, and procedure. Explanations of the symbol descriptors are given below.

1.1.1. Constant symbol descriptor

Constant symbol descriptors contain the attributes of named constants that appear in a program.

The example below illustrates the structure of the predeclared Boolean constant false.

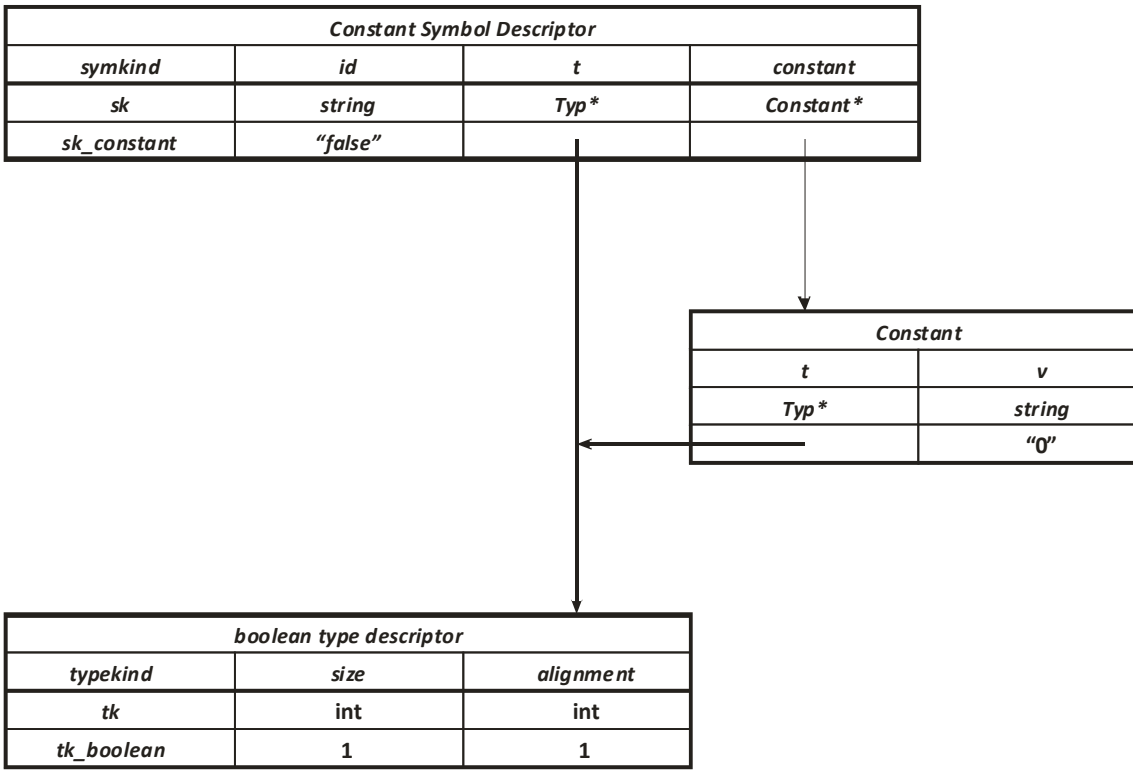

Figure 1. Constant symbol descriptor for the Boolean constant false.

1.1.2. Type symbol descriptor

Type symbol descriptors are used to associate the name of a type with its attributes. In Pascal *boolean*, *char*, *integer*, and *real* are names of the corresponding types. The names are assigned by the compiler and, surprisingly, are not reserve words. It is

possible but certainly not advisable[1] to assign these names to other types or constants. The facility to create named types is made available to the programmer in the **type** definition section of a Pascal program.

Example: type integer.

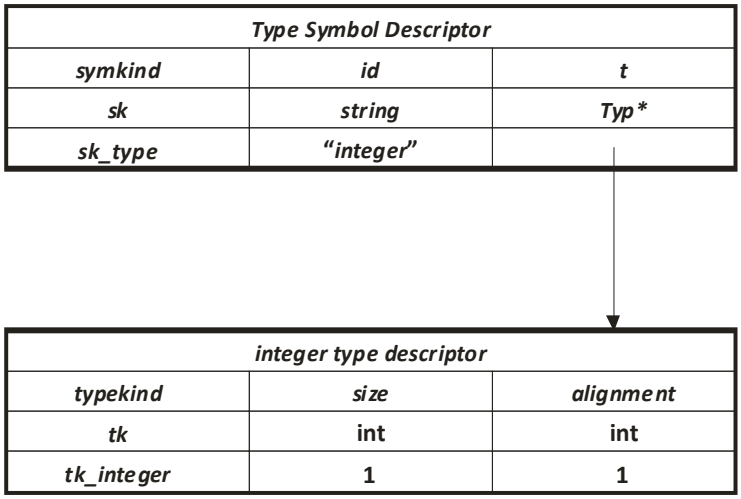

Figure 2. Type symbol descriptor for "integer"

Descriptors shown in Figures 1 and 2 are entered into the symbol table at lexical level zero (0) by the compiler.

[1] Once the names were assigned to other constants their original associations would not be accessible and one might not be able to declare any more variables of the type whose name was reassigned.

Example: **type** *index*=**1..10;**

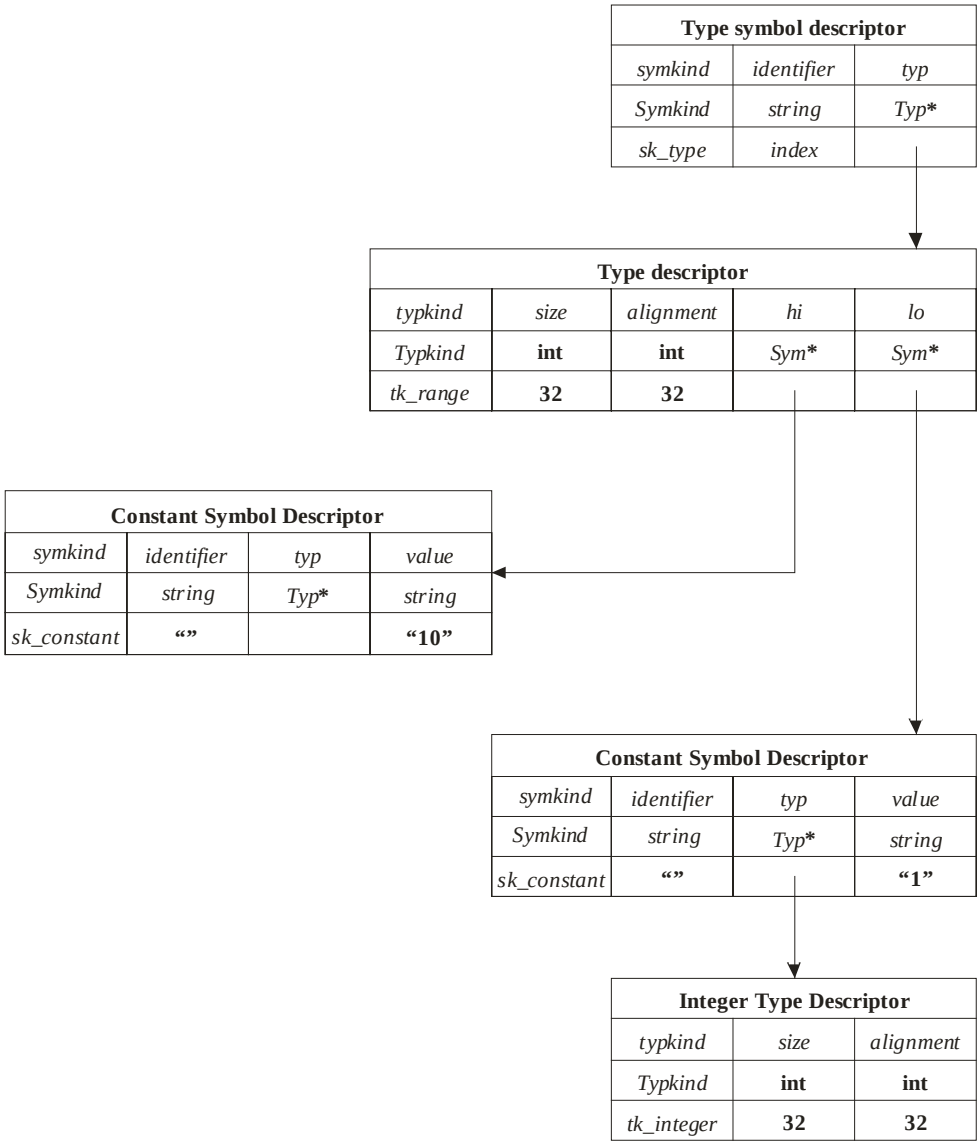

Figure 8. Type symbol descriptor for **type** *index*=**1..10;**

Figure 8 notes:

1. The identifier *index* is stored in the current namespace. What kind of identifier is *index*? It is the name of a type.
2. The type descriptor for identifier *index* is a sub-range. Sub-ranges are delimited by an upper bound (*hi*) and a lower bound (*lo*). The bounds are constants. In this case, the constants are unnamed integer constants.

1.1.3. Variable symbol descriptor

Variable symbol descriptors are employed to record the attributes of local variables, parameters, and fields in records. The common theme for a variable symbol descriptor is the relative address. A variable is allocated storage relative to the current activation record that contains storage for all variables local to the active subprogram. Parameters are treated exactly as variables. Storage for parameters is allocated from the current activation record. The address of a field is relative to

the start of the record and, therefore, can be managed in the same way as variables and parameters.

Example: **var** *c*:*char*;*r*:*real*;

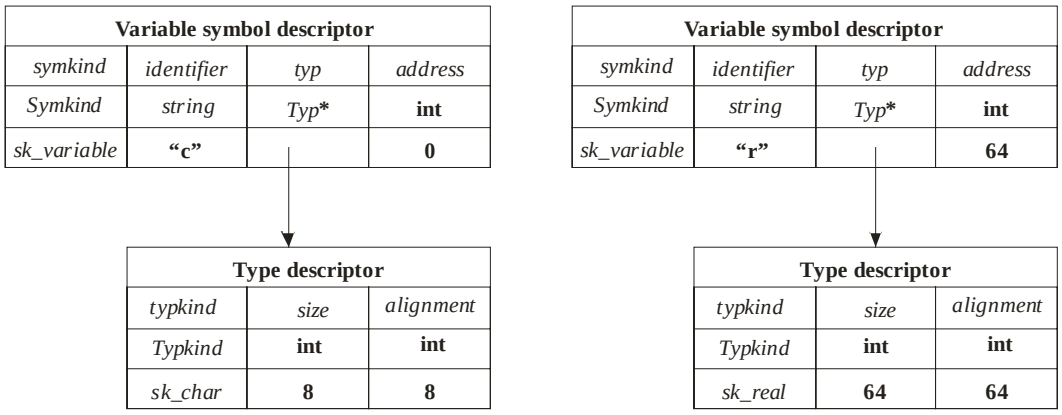

Figure 9. Symbol and type descriptors for the declarations **var** *c*:*char*;*r*:*real*;

Figure 9 notes:

1. First, a symbol descriptor for variable *c* is entered into the symbol table, then, the symbol descriptor for variable *r* is entered.
2. Type descriptors for type *char* and *real* exist in the namespace at lexical level zero (0): they were stored in the namespace by the compiler.
3. Since variable c was declared first, it is assigned the first available location at relative (bit) address zero (0)[2]. Next, variable *r* was assigned the next available location. Real variables can only be assigned locations in memory that begin on even multiples of 8-byte boundaries. The next available location for a real variable is not immediately after variable *c*, at location 8 bits (1 byte), but at 64 bits (8 bytes). The intervening space between variable *c* and variable *r* is not used.

1.1.4. Function and procedure symbol descriptors

Function and procedure symbol descriptors record the number and type of parameters and a function symbol descriptor records the return type. Type descriptors of the parameters are stored in a list. If the symbol descriptor records the attributes of a function the first type descriptor on the list is the return type as shown in figure 10.

Subprogram symbol descriptors also record an address. The address recorded is an *instruction* address not an address in an activation record. The instruction address is the address of the first instruction in the subprogram.

[2] In reality, zero is ***not*** the first available location for local variables. Storage for private subprogram control is allocated before local variables including the static and dynamic links, the old extreme pointer, the return address, and the return value. Storage for parameters and compiler temporaries is also allocated before local variables.

Example: **function** *adr***(***c***,***a***:***integer***):***integer***;**

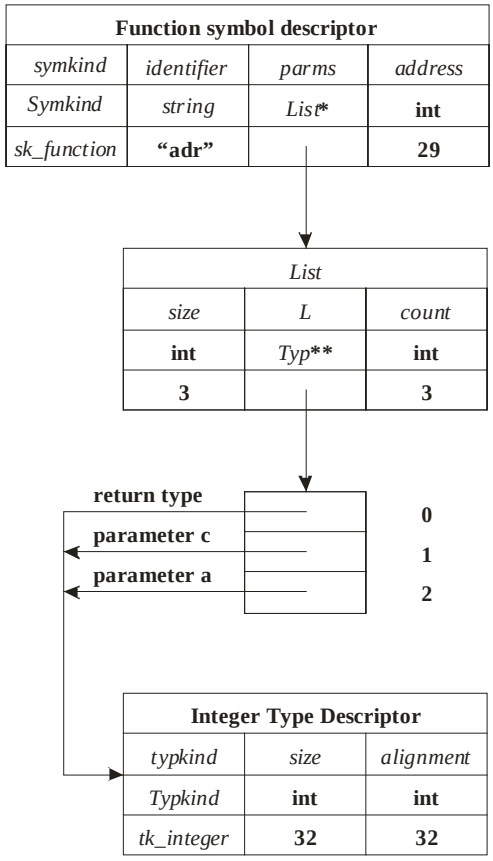

Figure 10. Symbol and type descriptors for **function** *adr***(***c***,***a***:***integer***):***integer***;**

Figure 10 notes:

1. Function *adr* has an *integer* return type. Element zero of the list of types records the return type of the function.
2. Function *adr* has two integer parameters, *c* and *a*. Elements one and two record the parameters types of the parameters.

Example: **procedure** *swap***(var** *m*,*w*:*real***);**

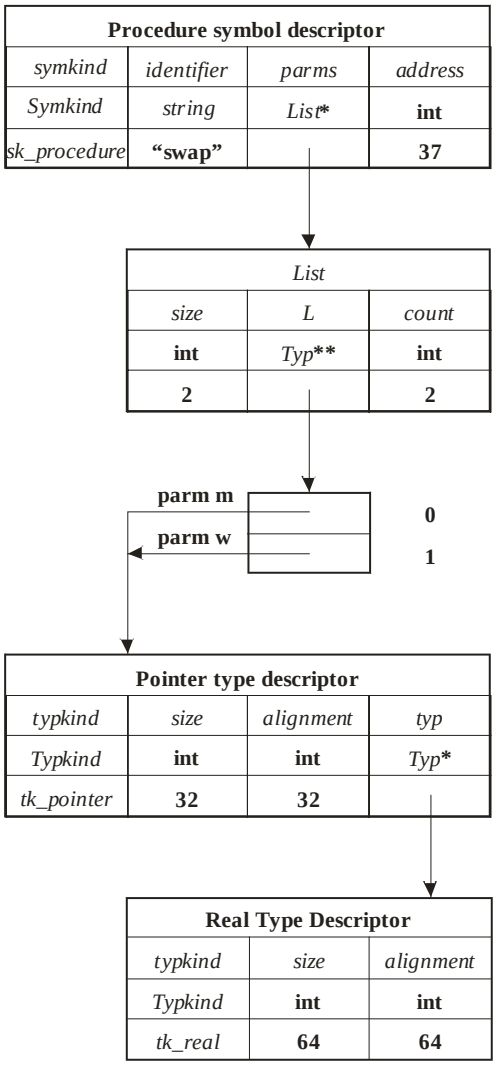

Figure 11. Symbol and type descriptors for **procedure** *swap***(var** *m*,*w*:*real***);**

Figure 11 notes:

1. Procedure *swap* has two parameters, *m* and *w*. Both parameters have type *real*.
2. Both parameters are reference parameters. The addresses of the values are passed to procedure *swap* rather than a copy of the actual values. The type of real reference parameters is *real******, a pointer to a *real*.

1.2. Type descriptors

Type descriptors record the attributes of a type. Types are very different and so are type descriptors. Symbol descriptors, because they describe identifiers have locality and must be stored in a namespace hierarchy that accurately reflects scope. Type descriptors, on the other hand, have no locality. They are not stored in namespaces. Access to types is via identifiers. Type descriptors are accessed by references stored in symbol descriptors.

Subsequent sections discuss kinds of type descriptors. Kinds of type descriptors include scalar type descriptors for character, integer, and real types. An enumerated type is used to record the attributes of type boolean in Pascal. Sub-range types are discussed next. Explanations and examples of type descriptors, for arrays, records, pointers, sets, and files are also given.

1.2.1. Character, integer, and real type descriptors

Character, integer, and real type descriptors have common attributes of *size* and *alignment*. The attribute size specifies the number of bits occupied by a variable or field of this type. The attribute alignment specifies where a variable of this type can be allocated in memory. An IEEE 754 double-precision binary floating point value can only be allocated on even multiples of 8-byte addresses. Thus, the alignment of an IEEE 754 double-precision binary floating point value (*real*) is 64, or 64 bits, to indicate that a variable of type real must begin on an 8-byte boundary. The units of *alignment*, like *size*, are bits.

The diagrams in figure 12 illustrate the symbol and type descriptors that are entered in the symbol table by the compiler at lexical level zero.

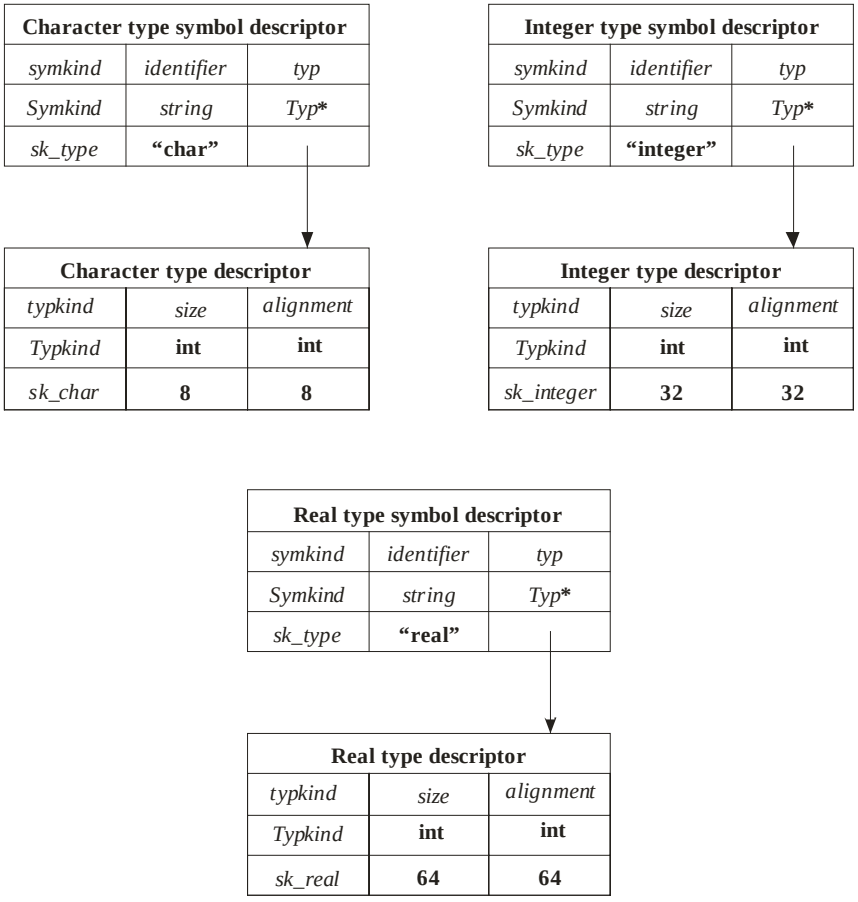

Figure 12. Symbol and type descriptors for types *char*, *integer*, and *real*.

1.2.2. Enumerated type descriptors and type boolean.

Pascal type boolean is an example of the use of an enumerated type descriptor. Pascal type boolean has two enumeration constants *false* and *true*. Constant symbol descriptors are used in the type descriptor for the enumeration type as shown in figure 13. Normally, the size and alignment of enumeration types and their constants is 32-bits. However, in this case, for type boolean, the size and alignment is 8-bits or one byte.

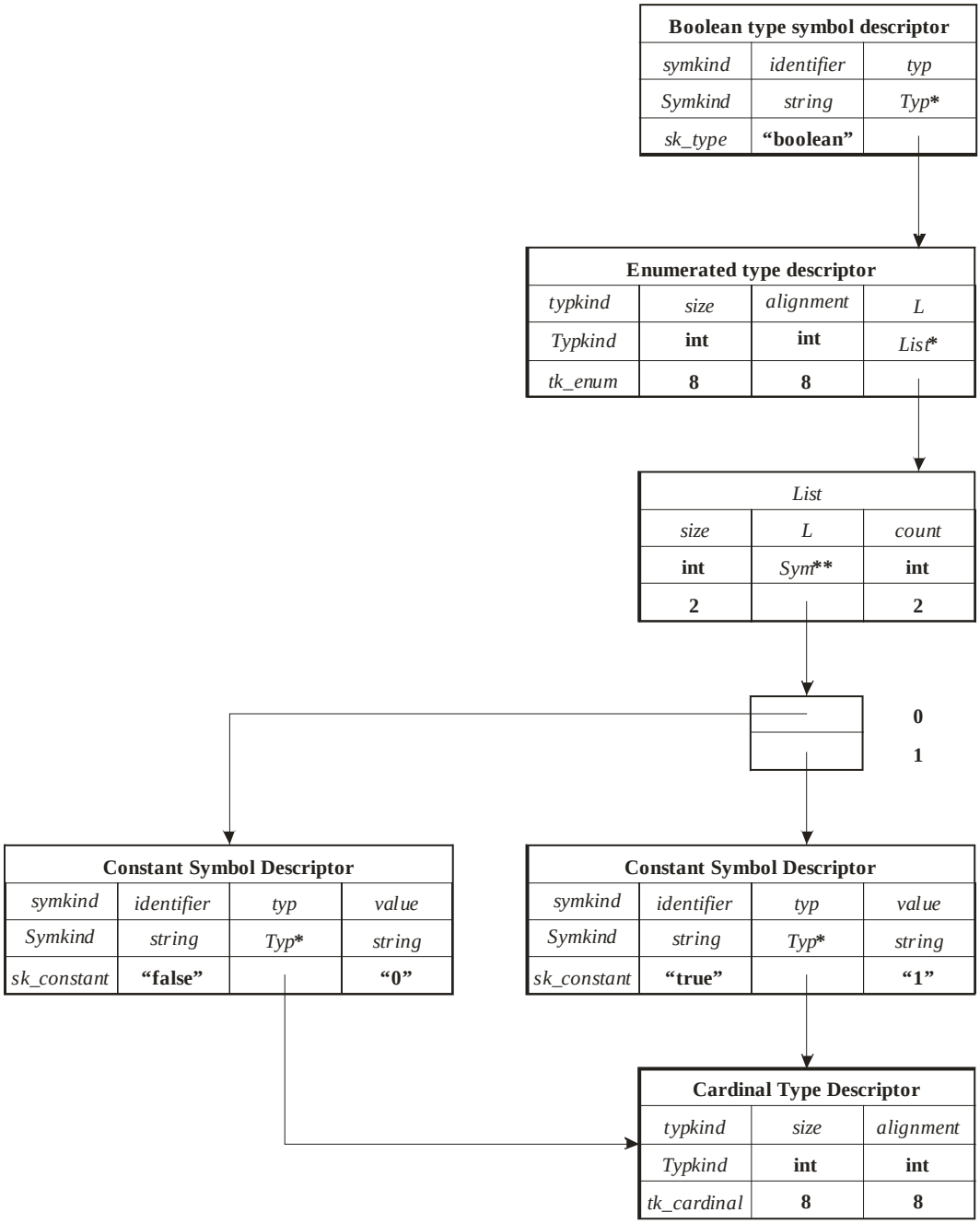

Figure 13. Symbol and type descriptors for type *boolean*.

1.2.3. **Range type descriptors.**

Range type descriptors are employed to record the attributes of a sequence of some type that is inherently ordered and discrete. Such types include cardinal (whole numbers), character, integer, and enumerated. The example below illustrates a range that includes the lower case letters.

Example: **type** *lowercase*=**'a' .. 'z';**

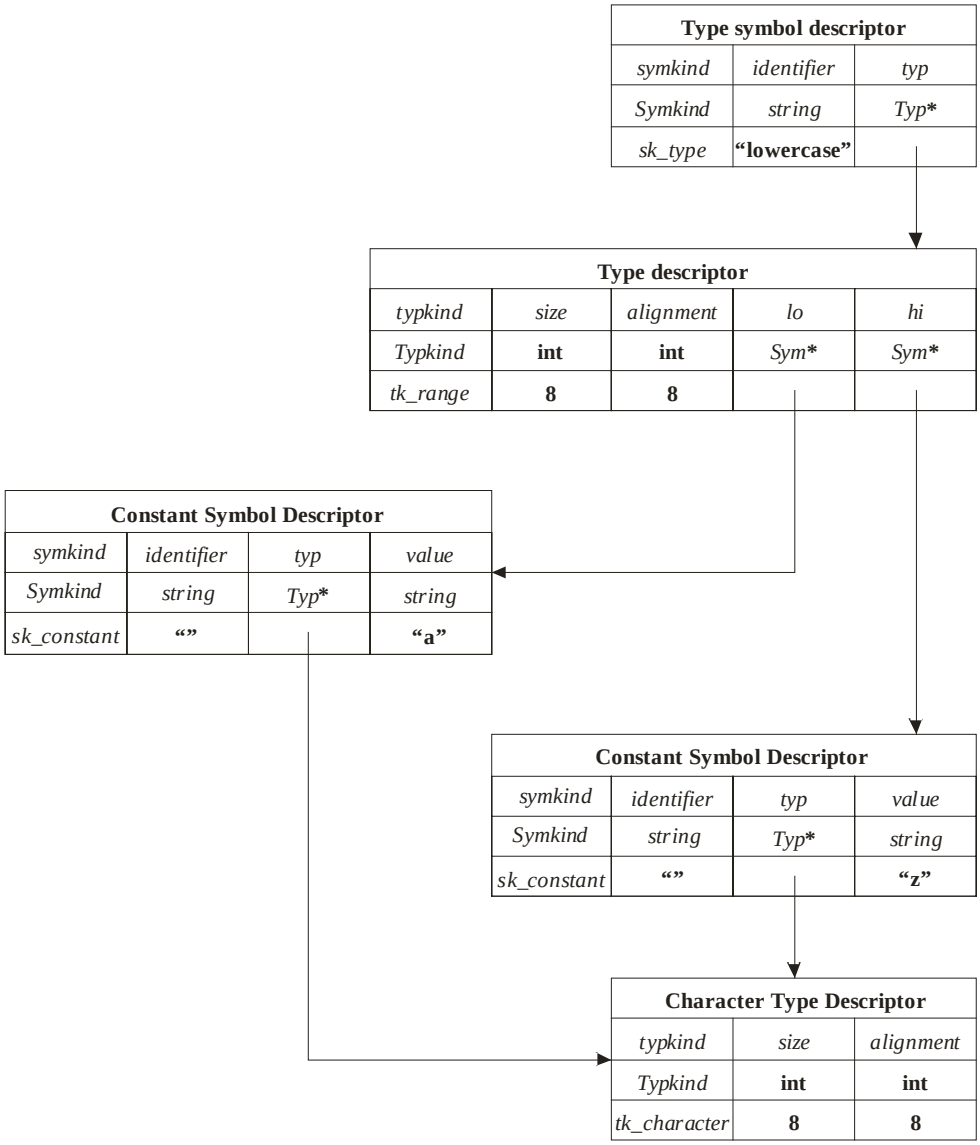

Figure 14. Range symbol and type descriptors for a used-defined type for lower-case letters.

1.2.4. **Array type descriptors.**

Array type descriptors reference subordinate types that describe the set of values that can be used as indexes for the array and the set of the values that can be used for the elements of the array. Index values are more restricted than element values. Index values must be discrete and must be a range of sequential values having integer representations. Acceptable index types include ranges of cardinal values, integer values, characters, boolean values, and enumerated values.

Element types can be used to construct multiply dimensioned arrays. Element types are unrestricted and can be an array type.

Two examples are given for arrays, 1) an array with a single dimension having a range of characters as the index type and 2) an array having two dimensions where ranges of cardinal values are employed as the index types.

Example: **type** *carry*=**array[‘a’..’e’] of** *integer*;

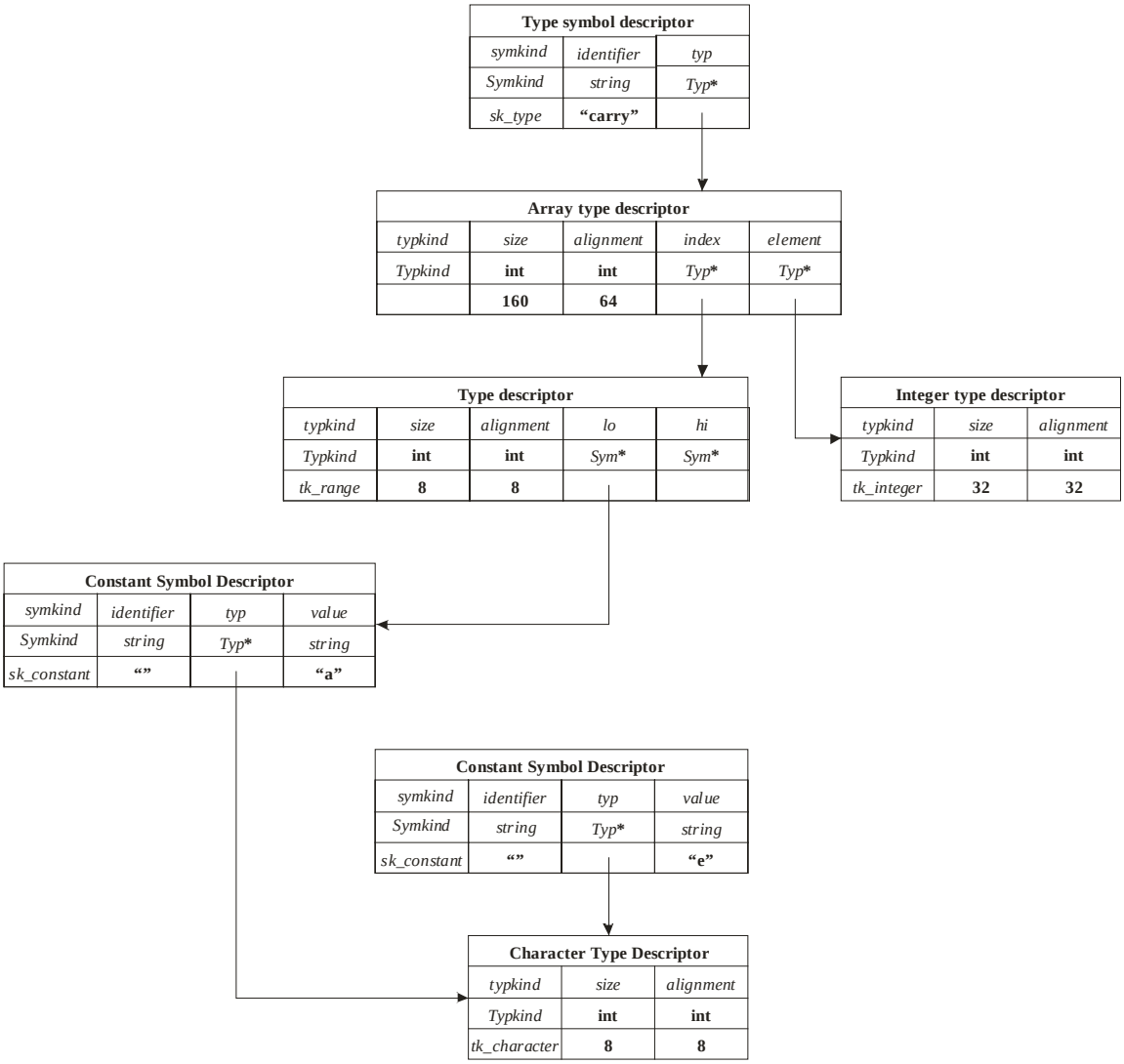

Figure 15. Symbol and type descriptors for the type definition **type** *carry*=**array[‘a’..’e’] of** *integer*.

Example: **type** *carry*=**array['a'..'e'] of** *integer*;

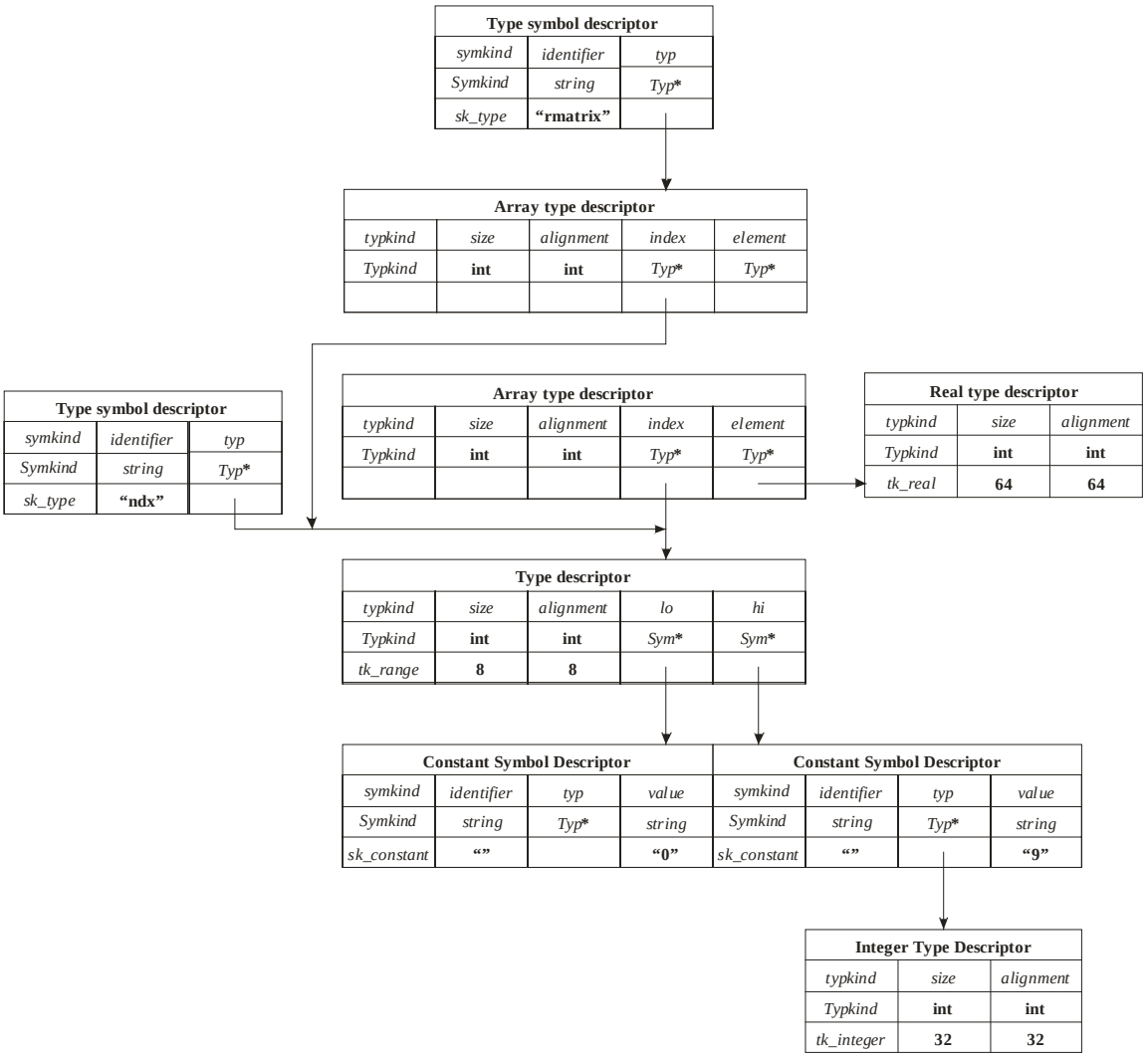

Figure 16. Symbol and type descriptors for the type definitions
type *ndx*=**0..9;** *rmatrix*=**array[***ndx*,*ndx***] of** *real***;**

1.2.5. **Record type descriptors.**

A record type descriptors contains a list of fields for the record. Each field is a variable symbol descriptor. Fields are recorded in the order in which they are declared.

Example: **type** *complex*=**record** *re*,*im*:*real* **end{complex};**

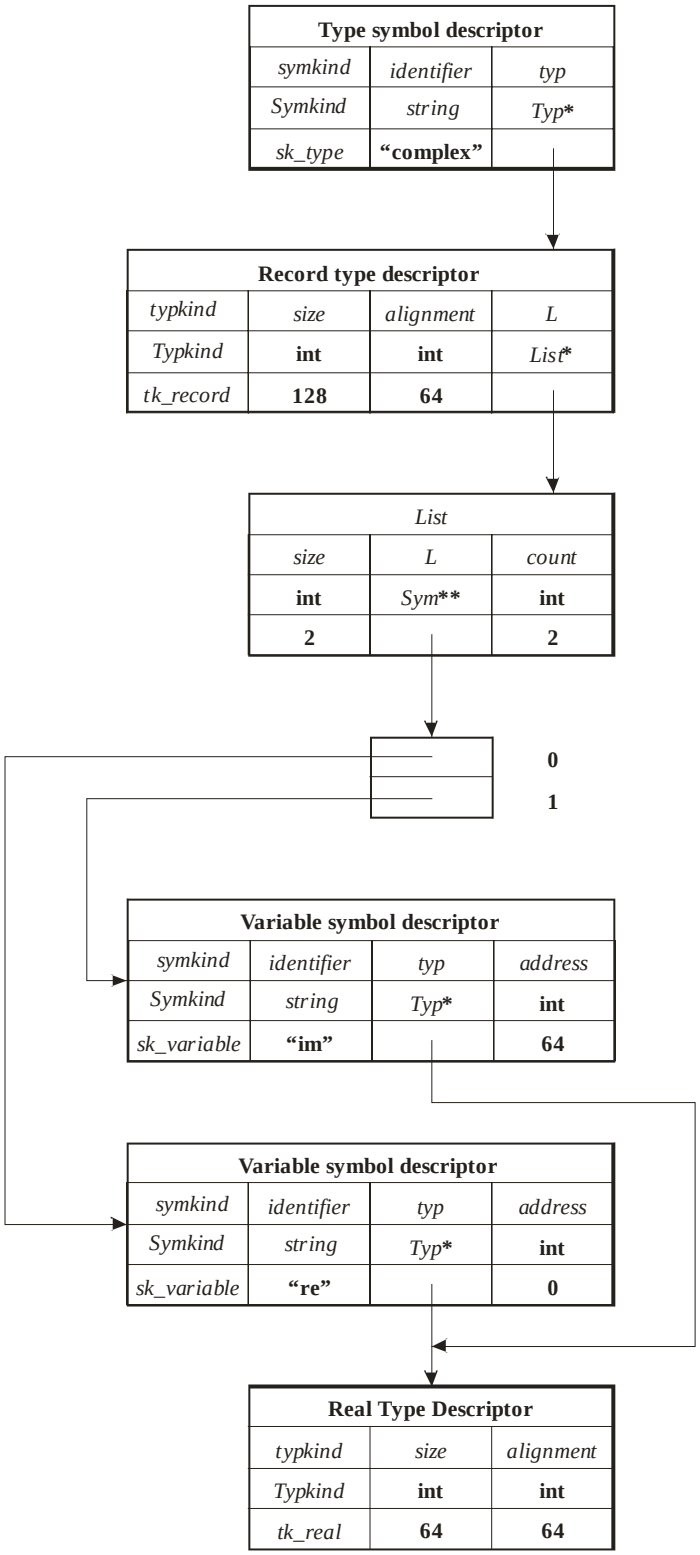

Figure 17. Symbol and type descriptors for the type definitions **type** *complex*=**record** *re*,*im*:*real* **end{complex};**

1.2.6. **Pointer type descriptors.**

Pointer type descriptors contain the attributes of a pointer or an address. An address occupies 32 bits and is aligned on 4-byte boundaries.

Example:

type

pelement=**^***element*;

element=**record** *prev:pelement; data:integer* **end{element};**

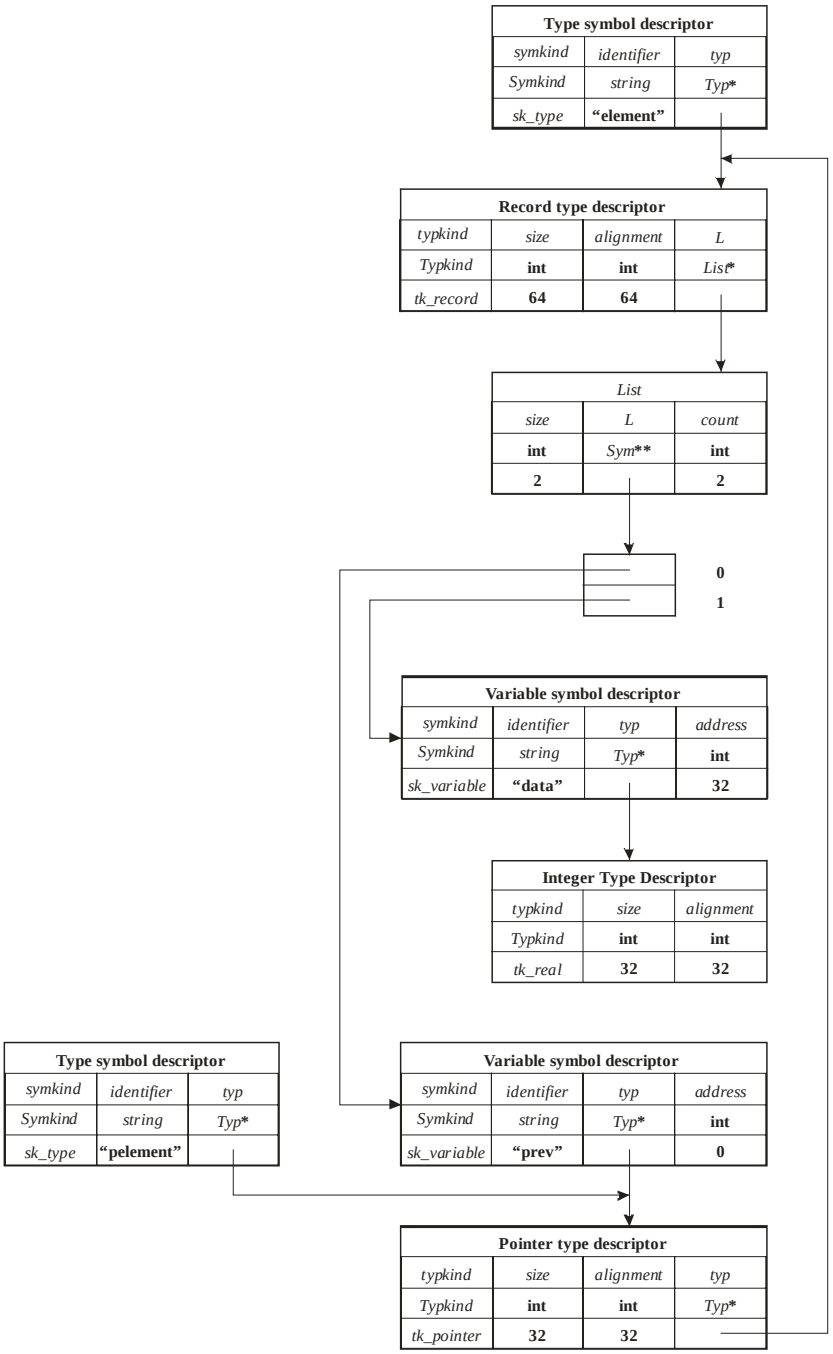

Figure 18. Symbol and type descriptors for the type definitions **type** *pelement*=**^***element*; *element*=**record** *prev:pelement; data:integer* **end{element};**

When pointers reference a record, the record is named as an incomplete type as in the example below. Type *element* is not yet defined in the definition of type *pelement*. Symbol table management must allow for the entry of types that are not yet defined but only named. Symbol table management must allow type symbols to be changed after their names are entered.

1.2.7. **Set type descriptors.**

Sets are implemented in an array of bits, one bit for each element in the set. If the bit is set to true, element is present in the set. If the bit is set to false, the set does not include that element.

In the example below, a variable of type *seasonset* occupies 64 bits. Bits 0, 1, 2, and 3 are used to represent the set. Bit 0 represents the presence or absence of the element *spring*. Bit 1 represents the element *summer*. In the same way, bits 2 and 3 represent elements *autumn* and *winter*.

The base type of a set must be discrete and its cardinality cannot exceed 64.
Example:
type *season*=**(***spring*,*summer*,*autumn*,winter**);** *seasonset*=**set of** *season*;

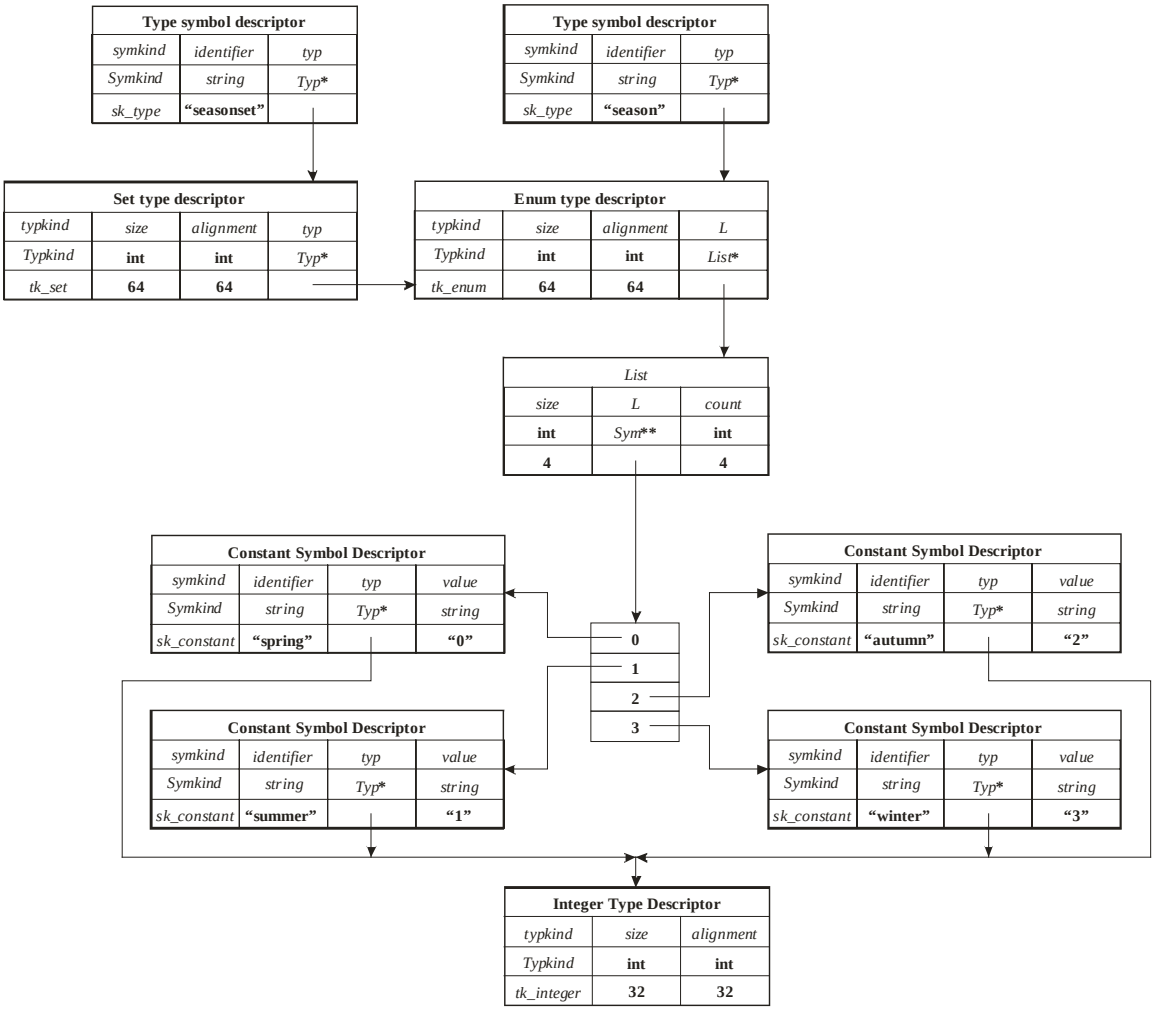

Figure 19. Symbol and type descriptors for the type definitions
type *season*=**(***spring*,*summer*,*autumn*,winter**);**

seasonset=**set of** *season;*;

1.2.8. **File type descriptors.**

Subset Pascal files are implemented as C++ file-streams and type descriptors are not defined in this document.