2. Data types

2.1 Introduction

The main purpose of this note is to explain how a Type Table can be incorporated into a compiler, to aid in dealing with the fact that programmers in a language like Modula-2 can create their own data types.

2.2 Categories of Types

The types with which a Modula-2 compiler has to deal are

1. Built-in types (BOOLEAN, CARDINAL, CHAR, etc.)
2. Enumerated types, e.g. Colour = (Red, Green, Blue)
3. Subrange [c1..c2] OF C
4. ARRAY C1,C2,... OF T
5. Record types
6. SET OF C
7. POINTER TO T
8. Procedure types, e.g. Handler = PROCEDURE (REAL): CHAR;

and maybe some others which I have forgotten. (By the way, I'm concentrating in this note on the general issues rather than on the precise language standards, so don't take the statements here as the final definitive answers. You should check the relevant "standards" to clear up the final details of the language rules.) In this list, T stands for *any* type, and C, C1, C2 stand for any countable types. The countable types are (from memory) all the built-in types except REAL, plus enumerated types and subranges.

The main parts of the compiler, and in particular the parts which handle variable designators, need to know this information in order to check for the legality of various constructs. For example, on meeting the construct A[x,y], the compiler needs to be able to check:

(a) that the type of A is an array type, since only arrays may have subscripts

(b) that the array must have two subscripts

(c) the types which x and y are required to have

(d) the final type of the combination A[x,y]

2.3 The Type Table

There is no limit on the number of types which a programmer can invent, other than the limit implicitly imposed by table overflow. (By the way, you should always check for table overflow when putting something into a table, and put out an appropriate error message if necessary. You should also be wary of using too many dynamic data structures, because it is much harder to put out a meaningful error message when dynamic memory overflows.) This suggests that every type should be designated internally by a cardinal number, and that the type specifications should be kept in an array TypeTable, so that TypeTable[j] gives the definition of type j. The first few type numbers should be reserved for the built-in types (BOOLEAN, INTEGER, etc.). This allows for checks like

```
IF type <= MaxBuiltinType THEN
     deal with the special cases associated with the built-in types
END (*IF*)
```

For each type other than the built-in types, the table entry needs to be a record of the form <category, other details> where the "category" field specifies one of the categories 1..8 listed at the beginning of Section 2.2.

The information needed in the "other details" part unfortunately varies from one category to another. You therefore need either a record with variants, or a pointer to another record which gives the details.

To illustrate what is needed, some of the cases will now be considered.

2.3.1 Pointer Types

This is the easy case. All you need to record, apart from the fact that this is a pointer type, is the type number of the type being pointed to.

2.3.2 Subranges

The information needed is <base type, lower bound, upper bound>.

2.3.3 Arrays

Here, you must store <element type, number of subscripts, type(s) of subscripts>. Since there can be an arbitrary number of subscripts, an appropriate way to store the information about the subscript types might be via a pointer to a linear list, in which each list element gives the type of one of the subscripts.

When compiling languages like Fortran, it is necessary to keep track of the subscript bounds. This is *not* necessary for Modula-2, because the bounds are implied by the subscript type − that is, the information about bounds is already in the table, in the entries for the subscript types.

2.3.4 Records

This is perhaps the trickiest case. I suggest that you use a linear list, in which each list element describes one field of the record. The information about a field should include the field name and the type of the field. When generating object code, it will be necessary to know the byte offset from the beginning of the record to the start of the field, so perhaps that should be stored here too. This raises a tricky point of principle: the byte offsets depend on the size of each element, so to be consistent you should include size information about every data type; but the sizes depend on the target machine, so in principle the front end of the compiler should not know about them.

The theoretically correct way to handle this problem is to have a separate table showing the size for every type, and showing the byte offsets of fields in the case of record types. I don't know whether any production compilers actually do it this way; I suspect that most of them ignore the problem and put the size information in the main type table.

The decision on what to do about this is up to you. It would be more elegant to have a separate "type size" table, but it's also more work. The main reason why it's more work is that practically every reference to a variable in the intermediate code requires a long expression giving the offset of that variable from the start of the frame in which it resides. (Notice that a stack frame is, in effect, an internally generated record.)

Although this discussion has not covered all possible cases, you should find it easy to look after the other cases.

2.4 Unnamed Types

Modula-2, like most similar languages, does not insist that you give a name to every type. For example, the programmer can write

```
VAR X: POINTER TO POINTER TO ARRAY [1..5] OF SET OF CHAR
```

in which several types have been defined without a name having been given to any of them. The compiler must handle this as if it had been written

```
TYPE
     t1 = [1..5];
     t2 = SET OF CHAR;
     t3 = ARRAY t1 OF t2;
     t4 = POINTER TO t3;
```

```
        t5 = POINTER TO t4;
VAR X: t5;
```

In other words, the compiler must invent new types as necessary (but it does not have to give them names − only numbers). In practice, there is nothing difficult about this; the compiler must simply make a new entry in the type table each time it encounters a type definition, whether it be named or unnamed.

Incidentally, it is worth noting that if the programmer writes

```
VAR  a: ARRAY[1..10] OF REAL;
     b: ARRAY[1..10] OF REAL;
```

then according to the rules of the language the variables a and b do *not* have the same type. This means that it is not necessary − or even desirable − for the compiler to check whether a new type definition is already in the type table.

2.5 Handling Ambiguities

Consider the declarations

```
VAR B: BYTE;  C: CARDINAL;  I: INTEGER;
```

followed by the executable statements

```
        B := 20;  C := 20;  I := 20;
```

This raises the question: what is the type of the constant 20? (Once you get down to the fine details of checking the language rules, you will find that it is important to answer questions like this.)

The solution to this problem is to define a new type "internal constant", and to manipulate the rules about assignment compatibility in order to make it legal to use these constants in places where you would expect to use them.

This still leaves some unresolved problems. For example, the result of ORD(0) should be 0 if 0 is considered to be a CARDINAL, but the result should be −MIN(INTEGER) if 0 is considered to be an INTEGER. In cases like this you just have to make an arbitrary decision, and document the fact that the language specifications are ambiguous.

2.6 Compatibility and Assignment Compatibility

A *strongly typed* language is one in which the assignment A := B is legal only if A and B have the same type; and where there are similar rules banning things like mixed mode arithmetic. Modula-2 is "almost strongly typed", in that types have to match most of the time but there are cases where types can be mixed. This means that you have to have, and enforce, rules about which types can be combined.

I don't have the rules about Modula-2 with me as I write this − you will have to look them up in a reference manual − but they are something along the following lines:

1. Two types are *equivalent* if they are the same type.
2. Two types are *compatible* if they are the same type, or if they are both subranges of the same type, or if ... (memory fails me here)
3. Two types are *assignment compatible* if they are compatible, or if ...

Incidentally, assignment compatibility is not a symmetric relationship. That is, if T1 is assignment compatible with T2 then it is not necessarily true that T2 is assignment compatible with T1.

The point of these definitions is to establish rules about when a statement is legal. For example, assignment compatibility is needed to make the assignment A := B legal. Assignment compatibility is also needed between a procedure parameter and the expression which appears in the procedure call. (That is in the case of a value parameter. For a VAR parameter, the types must be identical.) More subtly, assignment compatibility governs the rules about when mixed mode arithmetic is legal. For the expression (e1 + t), the new term t must be assignment compatible with e1.

Whenever two unequal (but assignment compatible) types are mixed, the compiler must insert a type conversion operation to convert the source operand into a value with the type of the destination operand.

Note that a type conversion is not the same thing as a type cast. A type cast is the operation T(X), where T is the name of a type and X is an expression of a different type. This simply means that the internal bit pattern which represents the value of X is re-interpreted as a value of type T, and no arithmetic is performed. (The result of a type cast is, of course, machine dependent.) In contrast, a type conversion operation performs some arithmetic consistent with the rules about how values can be converted from one type to another.

For example, the operation FLOAT(3) converts the CARDINAL number 3 to the REAL number 3.0, which has an entirely different internal bit pattern. The type cast operation REAL(3), if it works at all, will produce some machine-dependent value which has nothing in common with 3.0; and in any case it is legal only on machines for which the same number of bits is used to represent a CARDINAL and a REAL.