

# 7

## Data Types

### 7.3.3 With Statements

The Pascal with statement introduces a nested scope in which the fields of the named record become visible as if they were ordinary variables. The record is said to be *opened*. As shown in Figure ©3.19 (page ©32), a with statement can be implemented within the compiler by pushing an entry that represents the record type onto the symbol table scope stack.

#### EXAMPLE 7.115

Elliptical references in  
Cobol and PL/I

With statements are a formalization of the *elliptical references* of Cobol and PL/I, a language feature that permits portions of a fully qualified name to be omitted if no ambiguity results. In the Cobol equivalent of Example 7.46, name of elements(1) of chemical\_composition of ruby could probably be abbreviated name of elements(1) of ruby, since ruby is unlikely to have a field named elements within anything other than its chemical\_composition field. The rest of the reference is required, however, if there is another record of the same type as ruby in the current scope, and if the elements array within that type contains more than a single element. ■

#### EXAMPLE 7.116

Pascal with statement  
(reprise)

Elliptical references can be difficult to read, since they rely implicitly on the uniqueness of field names. A with statement specifies the elided information more explicitly, making misunderstandings less likely. Repeating example 7.46:

```
with ruby.chemical_composition.elements[1] do begin
  name := 'Al';
  atomic_number := 13;
  atomic_weight := 26.98154;
  metallic := true
end;
```

 ■

The with statements of Pascal still suffer from problems, however:

1. There is no easy way to manipulate fields of two records of the same type simultaneously (e.g., to copy some of the fields of one into corresponding

fields of the other). A `with` statement can be used to open one of the records, but not the other.

2. Naming conflicts arise if any of the fields of an opened record have the same name as local objects. Since the `with` statement is a nested scope, the local objects become temporarily inaccessible.
3. In a long `with` statement, or in nested `with` statements that open records of different types, the correspondence between field names and the records to which they belong can become unclear.

#### EXAMPLE 7.117

Modula-3 `with` statement

Modula-3 and Fortran 2003 address these problems by redefining the `with` statement in a more general form. Rather than opening a record, a Modula-3 `WITH` statement or Fortran `associate` construct introduces one or more aliases for complicated expressions. Recasting our example in the style of Modula-3, we can write:

```
WITH e = ruby.chemical_composition.elements[1] DO
    e.name := "Al";
    e.atomic_number := 13;
    e.atomic_weight := 26.98154;
    e.metallic := true;
END;
```

Here `e` is an alias for `ruby.chemical_composition.elements[1]`. The fields of the record are not *directly* visible, but they can be accessed easily, simply by prepending '`e.`' to their names. ■

#### EXAMPLE 7.118

Multiple-object `with` statements

To access more than one record at a time, one can write

```
WITH e = whatever, f = whatever DO
    e.field1 := f.field1;
    e.field3 := f.field3;
    e.field7 := f.field7;
END;
```

#### EXAMPLE 7.119

Non-record `with` statements

Modula-3 `WITH` statements and Fortran `associate` constructs can even be used to create aliases for objects other than records, e.g., to test and then use a complicated expression without writing it out twice:

### DESIGN & IMPLEMENTATION

#### With statements

A compiler generally implements `with` or `associate` statements by creating a hidden pointer to the opened or aliased record. All uses of the record inside the `with` statement access fields efficiently via offsets from the hidden pointer. Equivalent efficiency can usually be achieved without the `with` statement, but only if the compiler implements global common subexpression analysis, a non-trivial form of code improvement that we defer to Section 16.4.

```

WITH d = complicated_expression DO
  IF d # 0.0 THEN val := n/d ELSE val := 0.0 END;
END;

```

#### EXAMPLE 7.120

Emulating with in Scheme

```

(let ((d complicated_expression))
  (if (not (= d 0)) (/ n d) 0))

```

This code has roughly the same effect as the code in Example ©7.119. Example ©7.118, which copies fields of *f* into *e*, would require the use of non-functional language features.

#### EXAMPLE 7.121

Emulating with in C

In C one might write

```

{
  my_struct *e = &whatever;
  my_struct *f = &whatever;
  e->field1 = f->field1;
  e->field3 = f->field3;
  e->field7 = f->field7;
}

{
  double d = complicated_expression;
  val = (d ? n/d : 0);
}

```

This code depends on the ability of the C programmer to declare variables in nested blocks and to create pointers to nonheap objects. Pascal does not permit nested declarations; neither Pascal nor Modula-3 permits the nonheap pointers. Reference types in C++, which we will introduce in Section 8.3.1, can be used in place of pointers in the C example to produce an even closer approximation of the Modula-3 or Fortran syntax.

#### ✓ CHECK YOUR UNDERSTANDING

63. What is a *with* statement? What purpose does it serve?
64. What are *elliptical references*?
65. What are the limitations of *with* statements as realized in Pascal? How are these limitations overcome in Modula-3 and Fortran 2003? How are they avoided in languages like Scheme?
66. Explain how to emulate the behavior of *with* statements in languages like C.
67. Is a *with* statement purely a notational convenience, or does it have pragmatic implications as well?

