Dick Grune • Kees van Reeuwijk • Henri E. Bal
Ceriel J.H. Jacobs • Koen Langendoen

Modern Compiler Design

Second Edition

Springer

Dick Grune
Vrije Universiteit
Amsterdam, The Netherlands

Ceriel J.H. Jacobs
Vrije Universiteit
Amsterdam, The Netherlands

Kees van Reeuwijk
Vrije Universiteit
Amsterdam, The Netherlands

Koen Langendoen
Delft University of Technology
Delft, The Netherlands

Henri E. Bal
Vrije Universiteit
Amsterdam, The Netherlands

Additional material to this book can be downloaded from http://extras.springer.com.

ISBN 978-1-4614-4698-9 ISBN 978-1-4614-4699-6 (eBook)
DOI 10.1007/978-1-4614-4699-6
Springer New York Heidelberg Dordrecht London

Library of Congress Control Number: 2012941168

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Part V
From Abstract Syntax Tree to Intermediate Code

Chapter 11
Imperative and Object-Oriented Programs

In the previous chapters we have discussed general methods for performing lexical and syntax analysis, context handling, code generation, and memory management, while disregarding the programming paradigm from which the source program originated. In doing so, we have exploited the fact that much of compiler construction is independent of the source code paradigm. Still, a Java compiler differs considerably from a Prolog compiler: both require paradigm-specific techniques, in addition to general compiler construction techniques. We will explore these paradigm-specific techniques in the next four chapters. Figure 11.1 shows the numbers of the chapters and sections that contain material on the subjects of lexical and syntactic analysis, context handling, and code generation for programs in the four paradigms covered in this book. For each of the three subjects we have already considered the general methods; the paradigm-specific methods are covered as shown in the table.

		Imperative and object-oriented	Functional	Logic	Parallel and distributed
Lexical and syntactic analysis,	general:	Ch. 2 & 3			
	specific:	–			
Context handling,	general:	Ch. 4 & 5			
	identification & type checking:	Sect. 11.1			
	specific:	Ch. 11	Ch. 12	Ch. 13	Ch. 14
Code generation,	general:	Ch. 6–9			
	specific:	Ch. 11	Ch. 12	Ch. 13	Ch. 14

Fig. 11.1: Roadmap to paradigm-specific issues

There are hardly any paradigm-specific lexical and syntactic issues, so the general treatment in Chapters 2 and 3 suffices. One exception is the “offside rule” in some functional languages, in which the text layout plays a role in parsing; it is

D. Grune et al., *Modern Compiler Design*, DOI 10.1007/978-1-4614-4699-6_11, 511

described in Section 12.1.1. Two aspects of context handling are almost universal: identification and type checking; rather than dedicating a separate chapter to these, we cover them at the beginning of the present chapter, in Section 11.1. The paradigm-specific context manipulations are described where needed in Chapters 11 through 14.

The main differences in compilers for different paradigms lie in the kind of code they generate, and this is the primary reason for having a separate chapter for each of them. Most (but not all) compilers for imperative and object-oriented languages produce code at the level of an assembler or lower, whereas many (but by no means all) compilers for functional, logic, and parallel and distributed languages generate C or C++ code as their target code. The general code generating techniques have been discussed in Chapters 6 through 9. and the specific forms of code to be generated for the four paradigms are treated in Chapters 11 through 14.

Returning to the structure of the present chapter, we first discuss identifier/operator identification and type checking; operator identification requires special attention since in most languages operators are overloaded, and the context is needed to make the final identification. The rest of the chapter concerns code generation for the imperative and object-oriented languages. Since programs in these languages work by specifying direct instructions for the manipulation of explicit data, and since both the instructions and the data are relatively close to those of the hardware, the main issues in code generation for these languages are the mappings of source language data onto bytes and of source language statements to low-level instructions. These issues are covered in Sections 11.2 and 11.4, respectively. An intermediate position is occupied by the activation records and closures (Section 11.3). Activation records are data structures used by the run-time system to represent active ("running" or "suspended") routines; closures are representations of routines that allow some advanced operations on them, most prominently partial parameterization. Both combine data and flow of control.

Although functional, logic, and parallel/distributed programs often require data structures that are not direct mappings of programmer data and instruction sequences that do not derive directly from the program, much of the following discussion is still relevant for the more advanced paradigms.

Roadmap

11.1 Context handling

The context handling phase follows the lexical and syntactic phases. The latter are concerned with local phenomena and relate each item to its immediate neighbors only: for example, lexical analysis combines letters with adjacent letters and digits into identifiers, and syntax analysis combines adjacent identifiers and operators into expressions. Context handling, on the other hand, is concerned with long-distance relations: for example, it relates the type of a variable in a declaration to its use in an expression and relates the position of a label to its use in a goto statement. The connectors in these long-range relations are the identifiers: all applied occurrences of an identifier i in an expression are plugs which fit in one socket, the declaration for i, and from that socket they obtain information about type, lifetime, and so on.

The first task of the context phase is to take the annotated syntax tree and to find the defining occurrence of each applied occurrence of an identifier or operator in the program.

That done, the context phase can turn to its second task, performing the context checking on each node of the tree, as specified by the rules in the language definition. Most of the context rules in language manuals are concerned with type checking, and will, for example, specify that the condition expression in an if-statement must have type Boolean. Other rules may, for example, forbid a jump to a label inside a for-statement from a goto statement outside that for-statement.

Additionally, context handling may be used to perform the final steps in syntax checking. For example, the syntax of a language as given in the manual may forbid records (structs) with no fields, but it is probably more user-friendly to allow such records in the parser and to issue the error message “No fields found in record type declaration” during context handling than to exclude them syntactically and issue the message “Unexpected closing parenthesis”, or something similar, during parsing.

Context handling is also sometimes called **semantic checking**, which emphasizes the checking for correct meaning more than that for correct context use. Of course, correct meaning cannot be fully checked in any formal sense of the word, but often useful warnings about dubious meanings can be given. Examples are the detection of unused variables and routines, infinite loops, non-trivial expressions that have a trivial result (for example EXPR >= 0, where EXPR is an expression of type unsigned integer), and many others.

The problem with this is that much of this information becomes available more or less automatically during code optimization. Issuing the warnings in that phase, however, has the awkward property that the messages fail to appear when the program is compiled without optimizations. Duplicating part of the code optimization effort in the context handler just to give the proper messages is not attractive either. The reasonable compromise—doing the analysis and issuing the messages in the context phase and saving the results for the code generator— complicates the interfaces of all the phases between the context phase and the consumer of the information. Such are the worries of the compiler designer.

Context handling is one of the least "automated" phases of a compiler. For lexical analysis, parsing, and code generation many tools have been developed which generate efficient implementations from formal specifications. Attempts have been made to formalize and automate the context handling, as described in Chapter 4, but in practice this phase is usually hand-crafted, meaning that every restriction in the language specification has its own code to check it, and has its own potential for being wrong.

We will now first turn to two context handling issues that play a direct role in static correctness checking in most languages: identification and type checking.

11.1.1 Identification

At a conceptual level, identification is the process of finding the defining occurrence of a given applied occurrence of an identifier or operator. The **defining occurrence** of an identifier is the place of its main, and usually only, introduction. This introduction supplies information about the identifier: its kind (whether it is a constant, a variable, a module, etc.), its type, possibly an initial or fixed value, possible allocation properties, etc. The other occurrences of an identifier are its **applied occurrences** and are the consumers of this information. For example, in

```
INT month := 1;
WHILE month <= 12 DO
    print_string (month_name[month]);
    month := month + 1;
END WHILE;
```

the month in the first line is the defining occurrence, the others are applied occurrences.

This formulation of the problem does not cover all cases. Some languages allow **forward declarations**, which result in identifiers having more than one introduction. Other languages do not require identifiers to be introduced or declared at all; those languages have the information about the identifiers built in or the information is distributed over all the applied occurrences: type and other properties of an identifier follow from its form and/or use.

At the compiler construction level, these differences disappear: there is an information entry for each named item in the program, in which all information about this item is collected. It is the task of the identification process to connect all occurrences of an identifier to its proper entry. Information can then be inserted in and retrieved from this entry as desired. As explained in Section 2.11, the information base in which all the entries are kept is called the *symbol table* or *name list*.

Not all identifiers must be looked up in the same set of entries: for example, variables must be looked up among the local and global identifiers, field selectors must be looked up among the field selectors of a given type, etc. Each of these sets of entries defines a **name space**, and the syntactic position of an identifier in the program determines which name space it belongs in. The precise rules follow from the language manual, but usually identifiers in the syntactic positions of variable names, routine names, and some others live in one name space, the **general name space**, and field selectors live in special name spaces belonging to record types.

This implies that one can have a variable name i next to a field name i, without the two getting in each other's way, as shown in the C code fragment

```
struct one_int {
    int i;
} i;


    ...
    i.i = 3;
```

The first i in i.i is looked up in the general name space, leading to its identification as a variable of type struct one_int. The second i is then looked up in the special name space of the members of the type struct one_int.

Specific questions about the name spaces in a particular language must be answered by the language manual. Examples of such questions are whether labels live in the general name space or in a special label name space, and whether module names have a name space of their own. In principle any position that can be distinguished syntactically or contextually can have its own name space. C has three main name spaces, one containing the names of enums, structs, and unions, one containing the labels, and one containing all the rest; the latter includes variable identifiers, routine identifiers, type identifiers, and enumeration value identifiers. In addition, C has a name space for each struct and each union; these name spaces contain only the field selectors of the corresponding structs and unions.

11.1.1.1 Scopes

Some name spaces, especially those in block-structured languages, are scope-structured. These scopes work in stack fashion: there is a stack of scope elements, one for each scope entered, and all actions are performed on the top element(s). The rules are simple:

- a new empty scope element is stacked upon scope entry;
- declared identifiers are entered in the top scope element;
- applied identifiers are looked up in the scope elements from top to bottom; and
- the top element is removed upon scope exit, thus removing all declarations from that scope.

A naive implementation of a scope-structured name space is shown in Figure 11.2; on the left is the scope stack, on the right the linked lists of declaration information records, one for each name declared in each scope. The capital P in a record for a name stands for the properties attached to the name in the declaration.

Five levels of scope have been shown; they represent a possible constellation for a C program: level 0 is the library level, level 1 the program (routine declaration)

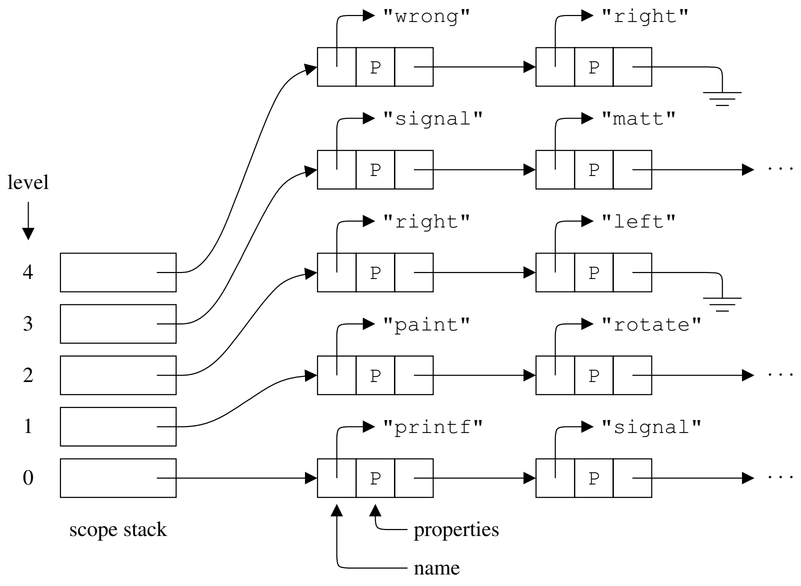


Fig. 11.2: A naive scope-structured symbol table

level, level 2 the formal parameter level, level 3 the local variable level and level 4 a subblock. Such a symbol table might result from the C code in Figure 11.3. This set-up allows easy insertion of new names and easy removal of entire scopes; identifiers can be found by performing a simple sequential search.

```
void rotate(double angle) {
    ...
}

void paint(int left, int right) {
    Shade matt, signal;

    ...
    {   Counter right, wrong;
        ...
    }
}
```

Fig. 11.3: C code leading to the symbol table in Figure 11.2

The organization shown in Figure 11.2 is simple and intuitive, but lacks two features essential for use in a practical compiler: it has no provisions for name spaces and it does not use the fast symbol table access discussed in Section 2.11. We have seen there that some identifier identification must already be done between the lexical scan and the parser, long before anything about scope and name spaces is known. This identification is based on the routine *Identify(Name)*, which yields a pointer to a record of type *IdfInfo*, which gives access to all information about the identifier *Name*. The speed of access is obtained using a hash table.

There are several ways to combine hash table identification, name spaces, and scopes in one implementation. We will discuss one possibility here, suitable for C-like languages; for a variation on this implementation, see Exercise 11.2.

Figures 11.4 and 11.5 together show an implementation that utilizes the easy access to names provided by a hash table, provides a fixed number of name spaces at the cost of a single field selection, and allows efficient scope stack operations. To avoid clutter, the diagrams depict a subset of the symbol table from Figure 11.2: only the names paint, signal, and right are shown.

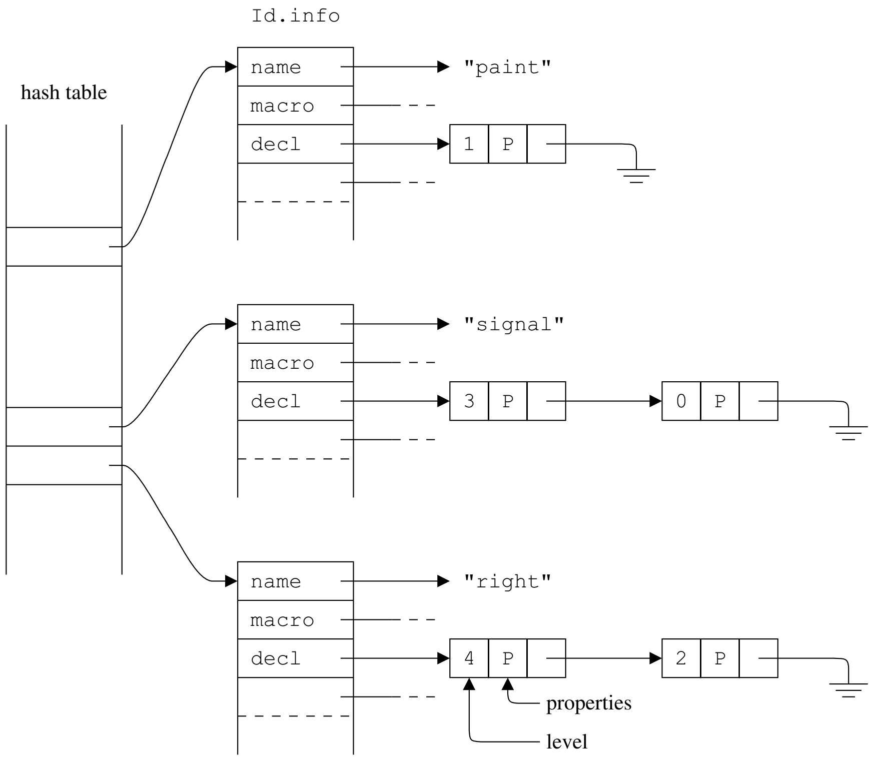


Fig. 11.4: A hash-table based symbol table

Figure 11.4 shows the identification part: the IdentificationInfo records are directly pointed to and accessed by the hash table. The name spaces are implemented as fields in the IdentificationInfo records, as follows. The record for identifier *I* starts with a pointer to the name *I*, for hash collision resolution and error reporting. Each following field represents the contents of a name space with respect to *I*. The first

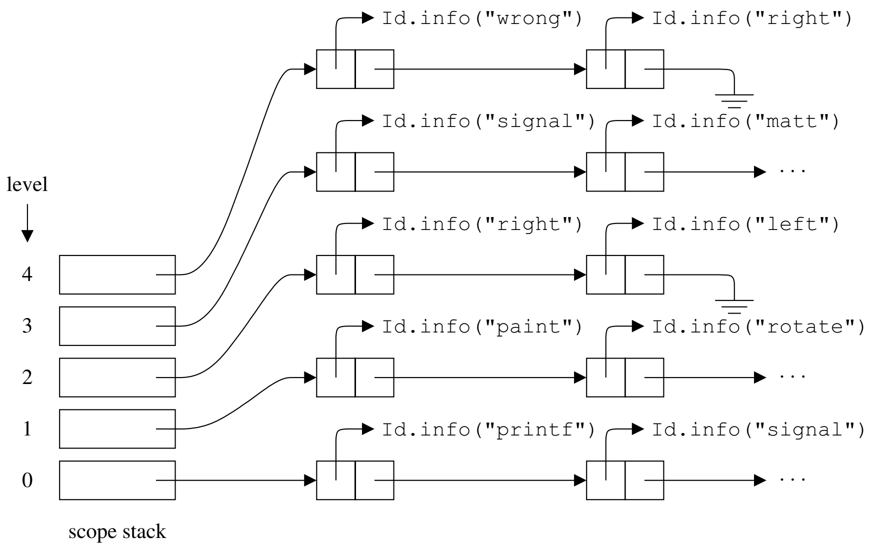


Fig. 11.5: Scope table for the hash-table based symbol table

field contains a pointer to a possible macro definition of *I*; in C, such a macro definition takes precedence over all other definitions. The decl field points to a stack of declarations of *I* in the general name space, implemented as a linked list of declaration information records. Each declaration information record is marked with its scope level, and the linked list is sorted on scope level. The top record is the identification sought: the declaration information of identifier *I* can be retrieved as *Identify(I)*.decl. This implements part of the scope stack and provides declaration identification in constant time. Further fields in the IdentificationInfo record provide access to information about *I* in other name spaces.

Figure 11.5 implements the rest of the scope stack: its structure mirrors the original implementation in Figure 11.2, except that the records pointed to by each scope entry contain pointers to the pertinent IdentificationInfo records rather than the declaration information records of the identifiers. The primary use of this structure is in removing declarations at scope exit in a narrow compiler. When the scope on top of the stack is to be removed, the top element on the stack points to a linked list of identifiers declared in that scope. Following that list, one can find and remove these declarations, as shown by the outline code in Figure 11.6. Deleting the list and the top element conclude the operation; the previous scope is restored automatically.

These data structures allow fast and easy addition of identifier declarations and the removal of entire scope information sets. Without these data structures, scope removal would involve scanning the entire symbol table, which may contain hundreds or thousands of identifiers, to find the ones that are declared on the level to be removed.

```
procedure RemoveTopmostScope ():
    LinkPointer ← ScopeStack [TopLevel];
    while LinkPointer ≠ NoLink:
        –– Get the next IdentificationInfo record:
        IdfPointer ← LinkPointer.idf_info;
        LinkPointer ← LinkPointer.next;
        –– Get its first DeclarationInfo record:
        DeclarationPointer ← IdfPointer.decl;
        –– Now DeclarationPointer.level = TopLevel
        –– Detach the first DeclarationInfo record:
        IdfPointer.decl ← DeclarationPointer.next;
        FreeRecordPointedAtBy (DeclarationPointer);
    Free (ScopeStack [TopLevel]);
    TopLevel ← TopLevel – 1;
```

Fig. 11.6: Outline code for removing declarations at scope exit

As we have seen, record and module declarations create named subscopes which themselves live in scopes. In idf.sel, idf is first looked up as an identifier in the identifier name space. This gives a pointer to a definition of idf, which among others holds a pointer to a record *T* describing the type of idf. *T*, which may be defined in a different (older) scope than idf itself, is then tested to see if it is a structured type. If it is not, the expression idf.sel tries to select from a type that has no selectors and an error message must be given.

In one possible implementation, shown in Figure 11.7, *T* provides a pointer to a list of records describing selectors, in which the selector sel can be looked up by sequential search. This will then lead to the type of the field. In Figure 11.7, it would be contained in property part *P* of the second record.

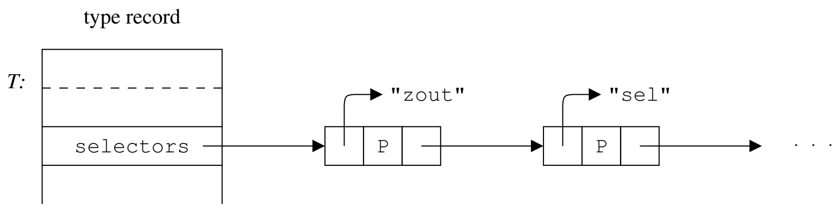


Fig. 11.7: Finding the type of a selector

Removing a level that holds definitions of structured types entails removing both the type definitions and their fields. This is not particularly difficult, but requires some care.

11.1.1.2 Overloading

If the source language manual states that all identifiers in a scope hide the identifiers with the same names in older (more outer) scopes, identification is relatively simple: the definition record pointed at by the decl field in Figure 11.4 provides the definition sought. In a language with overloading, however, identifiers do not hide all identifiers with the same name; for example, PUT(s: STRING) does not hide PUT(i: INTEGER) in Ada. Also, in many languages the operators are overloaded: the + in 3 + 5 is different from the one in 3.1 + 5.1, since the first has two integers as operands and the second operates on two floating point numbers.

The ambiguity caused by the overloading is resolved by considering the context in which the name to be identified occurs. There are two issues here. The first is that in the presence of overloading the identifier identification process comes up with a *set* of definitions, rather than with a single identification. These definitions are selected from the list of definitions supplied by the decl field in Figure 11.4. The rules for selecting such a set of definitions from the list of all definitions depend on the source language. The second issue then concerns the reduction of this set to a single definition. This reduction process is again based on language-specific rules that consider the context of the identifier. If, as a result of the reduction process, no applicable definition remains, the identifier is undefined; if more than one definition remains, the identifier is ambiguous; and if exactly one definition remains, it is the definition of the identifier in the given context.

Suppose, for example, that the compiler needs to identify the PUT in the Ada statement PUT("Hello"). In Ada, routine identifiers can be overloaded and the overloading has to be resolved by considering the types of the parameters. This is easily achieved by generalizing the access key of the symbol table. Normally, a symbol-table entry is uniquely identified by its scope and the symbol string. For a language with Ada-type overloading resolution we use the routine name *and the input parameters*, the **signature** of a routine. Thus PUT(STRING), PUT(INTEGER), and PUT(FILE_TYPE,STRING) can all have their own entry in the symbol table, because the signatures are different.

A considerable complication is that the types of the actual parameters that help determine the definition of a routine name can themselves be the result of overload resolution. The problem is even more severe when identification is combined with coercions; that situation is discussed in Section 11.1.2.3.

Now that we have seen how we can still identify names in the presence of overloading, we consider its consequences for removing the top scope upon block exit. The main change is that there can now be more than one definition with the same level in the definition record chains in Figure 11.4. It is easy to remove them all rather than just the first one, when visiting the *IdfInfo* record in the code in Figure 11.6.

11.1.1.3 Imported scopes

Some programming languages feature constructs that introduce copies of scopes from other places. Examples are the scope resolution operator in C++, the with-statement in Pascal and Modula-2, and the import declaration in Modula-2, Ada, and other languages with modules or packages. The details of these scope importers differ from language to language. In C++, a scope resolution operator x:: preceding a function definition introduces the field selectors of the class x in a scope around that function definition. The Pascal and Modula-2 construct WITH x DO ... END is similar: it starts a new scope in which the field selectors of the record x are available as identifiers in the general name space. The FROM module IMPORT declaration of Modula-2 and the use declaration of Ada, however, do not introduce a new scope but rather merge the new names into the present scope. Name clashes must be prevented in Modula-2 by explicitly selecting the imported names, and are avoided automatically in Ada by the overloading mechanism and the visibility rules associated with the use declaration. In all cases the result is that, with some identifiers, the programmer can just write sel, instead of idf.sel, or more importantly PUT("Text") instead of TEXT_IO.PUT("Text").

The obvious implementation of this feature is of course to retrieve the scope *S* to be imported, and to collect all names in it. Next, this collection is filtered as required, and a new scope is created and stacked if the semantics of the importing construct prescribes so. Each name *N* is then defined in the top-level scope in the usual fashion, with its properties derived from its entry in *S*. The time required by this technique is proportional to the number of names imported.

11.1.2 Type checking

The previous section has shown us that looking up an identifier leads to a declaration information record—a container for the properties of the object that the identifier identifies. The most basic and obvious, but therefore somewhat inconspicuous, property is the **kind** of the object—whether it is a constant, a variable, a routine, a type, a module, etc.; and the first step in checking the proper use of the identifier is checking this kind. Kind checking in itself is almost—but not entirely—trivial: for example, when the context requires a module identifier, it is simple to test if a module identifier has been supplied, using the declaration information. Some of the kinds, however, specify objects that involve values, for example constants, variables, and routines. These values belong to **types** —sets of values and operations— and their use is subject to often complicated and intricate rules, and has to be checked. This type checking is not at all trivial; we will therefore first concentrate on type checking, and address one issue in kind checking at the end of this section. For a comprehensive treatment of types in programming languages see Pierce [222].

Type checking is valuable because it can be seen as a simple but practical form of **formal verification** of a program: the programmer states that the value of variable is always element of a particular set, and the compiler verifies that this is so. Type systems are often intricate to get the most of this desirable property while still being comfortable for the programmer.

Type checking is involved in large parts of the annotated syntax tree. For example, the language rules usually specify which types can be combined with a certain operator; there are rules for formal and actual parameter types in a routine call; and assigning an expression value to a variable restricts the allowed types for the expression. Type information in a compiler has to be implemented in such a way that all these and many other checks can be performed conveniently.

Types are usually introduced by name through **type declarations**, as in

```
TYPE Int_Array = ARRAY [Integer 1..10] OF Integer;
```

which defines the type Int_Array, but they may also be introduced **anonymously**, as in the following variable declaration:

```
VAR a: ARRAY [Integer 1..10] OF Real;
```

When a type is introduced anonymously, it does not have a name in the source program, but for uniformity it needs one. Therefore, the compiler must use its imagination to invent a unique name, for example producing such internal names as #type01_in_line_35. So the above declaration of a is seen as an abbreviation of

```
TYPE #type01_in_line_35 = ARRAY [Integer 1..10] OF Integer;
VAR a: #type01_in_line_35;
```

Type declarations often refer to other type identifiers. In some languages they are allowed to refer to type identifiers that have not been declared yet; such references are called **forward type references**. Forward type references enable the user to define mutually recursive types, for example:

```
TYPE Ptr_List_Entry = POINTER TO List_Entry;
TYPE List_Entry =
    RECORD
        Element: Integer;
        Next: Ptr_List_Entry;
    END RECORD;
```

where the first occurrence of List_Entry is a forward type reference. However, forward type references also add some complications to the work of the compiler writer:

- The forward type references usually must be resolved. When, during processing, a forward type reference is met, the identifier in it is added to the symbol table, and marked as forward. Next, when a type declaration for this identifier is met, its symbol table entry is modified to represent the actual type.
- A check must be added for loose ends. At the end of a scope in which forward type references occur, the forward type references all must have been resolved if the language manual says so (and it usually does). This check can be implemented by checking all symbol table entries of the scope for being marked forward. If a forward type reference is met, it is a loose end, and must be reported.

- In some languages, a check must be added for circularity. The consequence of allowing forward type references is that the user can now write a circular type definition:

  ```
  TYPE x = y;
  TYPE y = x;
  ```

  which probably is illegal. We will see below how to deal with this problem. (C and family do not suffer from this problem since they do not allow forward references to types. They do support forward references to union/structure tags, which is sufficient in practice.)

11.1.2.1 The type table

All types in a compilation unit are collected in a **type table**, with a single entry for each type. For each type, the type table entry might, among others, contain the following:

- its type constructor ("basic", "record", "array", "pointer", and others);
- the size and alignment requirements of a variable of the type;
- the types of the components, if applicable.

Various information is being recorded for types:

- for a basic type: its precise type (integer, real, etc.);
- for a record type: the list of the record fields, with their names and types;
- for an array type: the number of dimensions, the index type(s), and the element type;
- for a pointer type: the referenced type;
- for other type constructors: the appropriate information.

The type table entry must contain all the type information required to perform the type checking and code generation. The exact content therefore depends on the source language at hand. The representation in the compiler, on the other hand, depends on the implementation language, the language in which the compiler is written. In an imperative implementation language, the representation of a type usually is a record with a variant part (or a union) for those items that depend on the type constructor. In an object-oriented implementation language, a type would be an object class with all fields and methods that all type constructors share, and each type constructor would have its own subclass, with extensions specific for the type constructor at hand.

To demonstrate how the type table is built, let us now consider a simple language with record and pointer as the only type constructors, and one built-in type: integer. Also, for the sake of simplicity, let us assume that there is only one, global, scope for type identifiers. In this language, the built-in type integer is a predefined identifier indicating an integer type. The compiler places the identifier in the symbol table and its type in the type table, before it starts processing the input. This results in

```
type table:        symbol table:
TYPE 0: INTEGER "integer": TYPE 0
```

where the type table is indexed by values of the form TYPE *n* (implemented as integers or pointers) and the symbol table is indexed by strings.

Because of possible forward type references, it is not possible to just build the type table as input is processed. One possible solution is to add identifier references to the type table, which must be resolved when all input has been processed. For example, let us process the following type declarations:

```
TYPE a = b;
TYPE b = POINTER TO a;
TYPE c = d;
TYPE d = c;
```

Processing the first type declaration, TYPE a = b;, might result in the following type table and symbol table:

```
type table:          symbol table:
TYPE 0: INTEGER      "integer": TYPE 0
TYPE 1: ID_REF "b"   "a": TYPE 1
                     "b": UNDEFINED_TYPE
```

Then, processing TYPE b = POINTER TO a; results in another type table entry for a (because references to identifiers have not been resolved yet). So, we add TYPE 2: ID_REF "a" to the type table and process TYPE b = POINTER TO TYPE 2. The UNDEFINED_TYPE symbol table entry for b is now resolved, and we obtain:

```
type table:                  symbol table:
TYPE 0: INTEGER              "integer": TYPE 0
TYPE 1: ID_REF "b"           "a": TYPE 1
TYPE 2: ID_REF "a"           "b": TYPE 3
TYPE 3: POINTER TO TYPE 2
```

The last two lines of the input are processed in the same way, and our type table and symbol table now look as follows:

```
type table:                  symbol table:
TYPE 0: INTEGER              "integer": TYPE 0
TYPE 1: ID_REF "b"           "a": TYPE 1
TYPE 2: ID_REF "a"           "b": TYPE 3
TYPE 3: POINTER TO TYPE 2    "c": TYPE 4
TYPE 4: ID_REF "d"           "d": TYPE 5
TYPE 5: ID_REF "c"
```

Now that the input has been processed, all identifier references must be resolved. There are two reasons for doing this, the more important one being cycle detection. The other reason is that it is convenient to have each type referred to by a single, unique, index in the type table.

The first step in resolving the identifier references is to replace the identifier references in the type table by type references: a reference to an entry in the symbol table which refers to an entry in the type table is replaced by a direct reference to this type table entry. Our modified type table now has the following contents:

```
type table:
TYPE 0: INTEGER
TYPE 1: TYPE 3
TYPE 2: TYPE 1
TYPE 3: POINTER TO TYPE 2
TYPE 4: TYPE 5
TYPE 5: TYPE 4
```

Now, cycles can be detected by the closure algorithm presented in Figure 11.8. The algorithm also resolves type references to type references: when it has finished, a type table entry can still be a TYPE, but the entry it then refers to is no longer a TYPE—unless it is a cyclic definition. The algorithm constructs a set *Cyclic* containing all type table entries describing a type with a cycle. After executing this algorithm, our type table looks as follows:

```
type table:
TYPE 0: INTEGER
TYPE 1: TYPE 3
TYPE 2: TYPE 3
TYPE 3: POINTER TO TYPE 2
TYPE 4: TYPE 4
TYPE 5: TYPE 4
```

and the *Cyclic* set contains both TYPE 4 and TYPE 5. For all members in the *Cyclic* set, an error message must be produced. Also, these type table entries can be replaced with a special ERRONEOUS_TYPE entry.

Data definitions:

1. T, a type table that has entries containing either a type description or a reference (TYPE) to a type table entry.
2. *Cyclic*, a set of type table entries.

Initializations:

Initialize the *Cyclic* set to empty.

Inference rules:

If there exists a TYPE type table entry t_1 in T and t_1 is not a member of *Cyclic*, let t_2 be the type table entry referred to by t_1.

1. If t_2 is t_1 then add t_1 to *Cyclic*.
2. If t_2 is a member of *Cyclic* then add t_1 to *Cyclic*.
3. If t_2 is again a TYPE type table entry, replace, in t_1, the reference to t_2 with the reference referred to by t_2.

Fig. 11.8: Closure algorithm for detecting cycles in type definitions

A good next step is to replace any type table reference TYPE n, where the corresponding type table entry n contains TYPE m, with a type table reference TYPE m. Note that there may be many of these references, in the symbol table, in the type table itself, as well as in other data structures in the compiler. This action converts our type table and symbol table to

```
type table:                      symbol table:
TYPE 0: INTEGER                  "integer": TYPE 0
TYPE 1: TYPE 3                   "a": TYPE 3
TYPE 2: TYPE 3                   "b": TYPE 3
TYPE 3: POINTER TO TYPE 3        "c": TYPE 4
TYPE 4: ERRONEOUS_TYPE           "d": TYPE 5
TYPE 5: ERRONEOUS_TYPE
```

and may make similar changes to other data structures in the compiler.

Another option is to have a function *ActualType()* in the compiler, which is called whenever a type table reference is required and which resolves a TYPE, for example:

```
function ActualType (TypeIndex) returning a type index:
    IF TypeTable [TypeIndex] is a TYPE reference:
        return TypeTable [TypeIndex].referredIndex;
    else –– TypeTable [TypeIndex] is a direct type:
        return TypeIndex;
```

This avoids the extra pass over the internal compiler data structures used by the first approach, but requires calls of *ActualType()* to be inserted in many places in the compiler code, which may be error prone.

For a different approach to scope table management see Assmann [20].

11.1.2.2 Type equivalence

When type checking an expression or the formal and actual parameters of a routine call, often two types must be compared to see if they match in the given context. For example, a routine may be declared to have a parameter of a floating point type, and a call to this routine may have an integer parameter. The compiler has to detect this situation, and produce an error message if the language rules do not allow this. Which combinations are allowed in which context is specified by the language manual. For example, the routine call of the example above is allowed in (ANSI) C, but not in Modula-2.

An important part of comparing two types concerns the notion of **type equivalence**. When two types are equivalent, values of these types have the same representations: one can be used where the other is required, and vice versa. The language manual, again, specifies when two types are equivalent. There are two kinds of type equivalence, structural equivalence and name equivalence. Since structural equivalence is tricky to define and implement well, modern languages tend to use name equivalence whenever possible. However, in some cases, for example for array equivalence, using structural equivalence makes for a much more useful language.

Name equivalence Two types are **name-equivalent** when they have the same name. Note that this requires every type to have a name (either user-declared or anonymously assigned by the compiler). In a language with name equivalence, the two types t1 and t2 in

```
TYPE t1 = ARRAY [Integer] OF Integer;
TYPE t2 = ARRAY [Integer] OF Integer;
```

are not equivalent since they have different generated names, but the following two types are:

```
TYPE t3 = ARRAY [Integer] OF Integer;
TYPE t4 = t3;
```

Implementing a name equivalence check is easy: name-equivalent types have the same index in the type table, once the ID_REF entries as described above have been resolved.

Structural equivalence Two types are **structurally equivalent** when variables of these types can assume the same values (when the two types have the same structure) and allow the same operations. This makes the types

```
TYPE t5 = RECORD c: Integer; p: POINTER TO t5; END RECORD;
TYPE t6 = RECORD c: Integer; p: POINTER TO t6; END RECORD;
TYPE t7 =
    RECORD
        c: Integer;
        p: POINTER TO
            RECORD
                c: Integer;
                p: POINTER TO t5;
            END RECORD;
    END RECORD;
```

all equivalent.

Testing for structural equivalence is difficult. The algorithm basically works as follows: first, all types are placed into a single equivalence class. Then, repeatedly, attempts are made to split each equivalence class E into two equivalence classes E_1 and E_2, by selecting a single type T as a pivot from E and putting all types that can be shown to be not equivalent to T in E_1 and all types that cannot be shown to be not equivalent to T in E_2, according to certain rules.

11.1.2.3 Coercions

Type equivalence is only a building block in, not the complete answer to, type checking. In type checking, we want to answer the following question: if we expect a value of type T_1 at a particular position in the program, but we find a value of type T_2, is that acceptable? In this context, the type expected is sometimes called the **a posteriori type** and the type found the **a priori type**, but the terms "type expected" and "type found" are more intuitive and less error-prone. If T_1 and T_2 are equivalent, the type check certainly succeeds; if they are not equivalent the rules are language-dependent. For example, if a is a variable of type real, we expect a value of type real in the right-hand side of an assignment to a. However, the assignment a:=5 may have to be dealt with as well. Some languages do not allow any type mismatch, others require the compiler to insert a data conversion from integer to real. Such an implicit data and type conversion is called a **coercion**.

Exactly which coercions the compiler can insert depends on the language. In general, more than one coercion may have to be applied. For example, in the assignment xx := 5 where xx is of type complex, a coercion from integer to real and then a coercion from real to complex may be required. In many languages, the possible coercions also depend on the context. For example, in C, the context "right-hand side of an assignment expression" differs from the context "operand of a binary operator". In the former context, a coercion from real to integer is allowed, in the latter it is not: in

```
int i = 2.7;
```

the 2.7 is coerced to integer (which in C involves truncation, so the value becomes 2), and in

```
1 + 2.7
```

the 1 is coerced to real rather than the 2.7 to integer, to allow the + to be identified as a floating point addition.

As the above example already shows, the presence of coercions complicates operator and routine identification: operand and result types may need to be coerced to other types, before a matching identification can be found. Also, unrestricted application of coercions may lead to ambiguities. For example, the + in the expression 2+3 indicates integer addition, but if a coercion of the operands from integer to floating point is allowed, it could also indicate floating point addition.

Finding the proper coercions in an AST, for a language with arbitrary sets of types, contexts, and coercions, is a very difficult problem, for which no solution has yet been found. The approach presented here is similar to the one used in Section 11.1.1.2 to handle overloading of identifiers, and works for moderately complicated coercion rules. It is based on two closure algorithms, to be applied in succession. The first finds all types in each node that might play a role, the second crosses out all inapplicable types; the details depend on the source language rules. If in the end any type set in a node is empty, no identification could be made, and if any of the type sets contains more than one element, an ambiguity has been detected.

The closure algorithms are given in Figures 11.9 and 11.10. The sets with which the type sets of the nodes are initialized can be built in (as for a leaf 3.14) or supplied by the identification mechanism. Note that even the leaf nodes get type sets, since they too may be overloaded (for example enumeration values in Ada), and/or coerced. The inference rule in Figure 11.9 adds all types reachable by coercions; usually there are very few of these. The inference rules in Figure 11.10 then remove all types that are upward or downward incompatible. Note that the algorithm only determines the types; applying the necessary modifications to the expressions requires separate code.

As said before, the algorithm presented here is not the answer to all identification problems in the presence of overloading and coercions. In fact, as it stands, it is not even capable of handling operator identification in C. The basic problem is mainly one of language design and can be sketched as follows.

Data definitions:

1. S of a node, a variable type set attached to that node in the expression.
2. C of a node, a non-variable context associated with that node.

Initializations:

The type set S of each operator node contains the result types of all identifications of the operator in it; the type set S of each leaf node contains the types of all identifications of the node. The context C of a node derives from the language manual.

Inference rules:

For each node N with context C, if its type set S contains a type T_1 which the context C allows to be coerced to a type T_2, T_2 must also be present in S.

Fig. 11.9: The closure algorithm for identification in the presence of overloading and coercions, phase 1

Data definitions:

S of a node, a type set attached to that node in the expression.

Initializations:

Let the type set S of each node be filled by the algorithm of Figure 11.9.

Inference rules:

1. For each operator node N with type set S, if S contains a type T such that there is no operator identified in N that results in T and has operands T_1 and T_2 such that T_1 is in the type set of the left operand of N and T_2 is in the type set of the right operand of N, T is removed from S.
2. For each operand node N with type set S, if S contains a type T that is not compatible with at least one type of the operator that works on N, T is removed from S.

Fig. 11.10: The closure algorithm for identification in the presence of overloading and coercions, phase 2

Suppose a language has two types, int and real, and two +-operators, with types (int, int) $\rightarrow$ int and (real, real) $\rightarrow$ real. To accommodate expressions like 3.14 + 5 the language has to allow a coercion from int to real in operand context, to reach the expression 3.14 + (real)5, which identifies the second +. But having such a coercion in operand context makes the expression 3 + 5 ambiguous, since it allows both 3 + 5 to identify the first + and (real)3 + (real)5 to identify the second +.

This problem in language design is solved by having rules like “In operand context, a value can only be coerced to real if the other operand is real without using a coercion”. Such rules make the coercions allowed in one syntactic position dependent on what is found in another syntactic position. This kind of longer-range relationship is not supported by the algorithm presented here and has to be coded separately, either as a third scan over the expression or as ad-hoc code in the inference rule in Figure 11.10.

Another potential problem with the algorithm is that phase 1 will not terminate for certain coercion rules. This happens, for example, with the coercion rules of Algol 68, where the inference rule in Figure 11.9 will continue forever to require new types. Fortunately very few programming languages have such a complicated coercion mechanism as Algol 68, so this phenomenon is usually not a problem.

11.1.2.4 Casts and conversions

Some languages allow the use of casts. A **cast** specifies the required type explicitly. For the compiler, it just introduces a different context, with usually different and stronger coercion rules. A cast differs from an explicit **conversion** in that it still uses the coercion system of the language. In contrast, an explicit conversion is a function which transforms data of one type to that of another type, and the types of its operands and result are subject to the normal source language type rules. Of course, the function could be a built-in function of the language, in which case the compiler must know about it. On the other hand, it could just be a library function.

11.1.2.5 Kind checking

With one exception, kind checking is trivial, as explained at the start of this section. The exception concerns constants and variables, and the complication arises from the fact that the actual kinds we are concerned with are locations and values rather than constants and variables. We will now examine the relationships between these notions.

In the assignment *destination*:=*source*, we expect a location on the left and a value on the right. If we adhere to the idea that the assignment operator, like all operators, requires values as input, we expect a value which is the address of a location on the left and a (normal) value on the right. Based on the topology of the assignment statement, the first is called an **lvalue** (pronounced “el-value”) and the second is called an **rvalue**.

In the assignment p := q;, in which p and q are variables, we expect an lvalue for p and an rvalue for q. Since p is a variable, it has a location and its address is the lvalue; one even says that “a variable is an lvalue”. On the other hand, q is an lvalue too, but in its position an rvalue is required. The kind checking system solves this by inserting a coercion which retrieves the contents of the location addressed by an lvalue. This coercion is similar to the dereferencing explained below, but is less conspicuous because it is usually incorporated in the machine instructions. Although the AST for the assignment p := q;, as amended by kind checking, contains a dereference node (Figure 11.11, in which the arrows show the dependencies), this node is not reflected explicitly in the corresponding code sequence:

```
Load_Mem q,R1
Store_Reg  R1,p
```

It is hidden inside the machine instruction Load_Mem, as explained in the introduction to register machines at the beginning of Section 7.5.2.

We are now in a position to formulate the kind checking rules for lvalues and rvalues. The table in Figure 11.12 shows the basic rules; a – indicates that no action

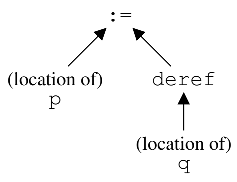


Fig. 11.11: AST for p := q with explicit deref

is required. The lvalue/rvalue attribute propagates bottom-up in the AST of an expression, according to language-dependent, but usually obvious, rules. Some of the rules are stated in the table in Figure 11.13, for C or a similar language; *V* stands for lvalue or rvalue.

The combined rules state, for example, that a[1] is an lvalue if a is a variable; type checking then tells us that a must be an array. This means that a[1] can be used as the destination in an assignment, since an lvalue is expected in that position. Suppose, however, that the expression a[1] is used as an index, for example in a[a[1]]; here an rvalue is required, so the lvalue needs to be dereferenced. On the other hand, 3 := 4 is erroneous, since 3 is an rvalue, and an lvalue is required; and so is &x := 7, for the same reason.

		Expected	
		lvalue	rvalue
Found	lvalue	--	deref
	rvalue	error	--

Fig. 11.12: Basic checking rules for lvalues and rvalues

Expression construct	Resulting kind
constant	rvalue
identifier (variable)	lvalue
identifier (otherwise)	rvalue
&lvalue	rvalue
*rvalue	lvalue
V[rvalue]	*V*
V.selector	*V*
rvalue + rvalue	rvalue
lvalue := rvalue	rvalue

Fig. 11.13: lvalue/rvalue requirements and results of some expression constructs

11.1.3 Discussion

The main problem in context handling for imperative and object-oriented languages is the identification of types, operators and identifiers. The problem is complicated by forward type references, type equivalence rules, routine and operator identification rules, overloading of identifiers and operators, and context-dependent coercions. No hard-and-fast general algorithm is available, but for almost all practical languages the problem can be solved by some form of inference rules working on type sets in the nodes of the expressions, and for many languages much simpler approaches involving only one type per node are possible.

11.2 Source language data representation and handling

In this section we discuss some of the data structures that represent source language data at run time, and the run-time manipulation needed to deal with the source language operations on these data. In the source language, a data item has a type, which may be a basic, built-in type of the language, or a constructed type, built using one of the type constructors in the language. The target language data types are usually limited to single bytes, integers of various sizes, address representations, and floating point numbers of several sizes.

Every source language data type is mapped to a particular combination of target language data types, and the run-time representation of a source language data item is the result of the application of this mapping. It is the task of the compiler writer to create such a mapping. We assume here that the target language has the common arithmetic, comparison, and copy operations, and has a byte-addressable memory; this assumption is almost universally justified.

11.2.1 Basic types

The usual **basic types** in a source language are characters, integers of several sizes, and floating point numbers of several sizes. The source language operations on these typically are arithmetic operations, assignment and comparison. The arithmetic operations include addition, subtraction, multiplication, division, and remainder. All these can be mapped directly to the target language data types and operations. Often the same instructions are used for signed and unsigned arithmetic, but the comparison instructions usually differ. Traditionally, characters were mapped to single bytes using the ASCII encoding, but many modern source languages support Unicode [277], and require two or four bytes to represent all possible character values.

Some source languages also have a **void type**, corresponding to no value at all. In some other languages a void type is present, but only implicitly. Representation of the void type in the target language is usually trivial or not necessary at all.

Floating-point numbers are in principle amenable to all the usual expression optimizations and simplifications, but due to their complicated arithmetic rules a number of obvious optimizations are incorrect. For example, the “obvious” simplification of v*0 to 0.0 fails, because for v = ∞ the result of the multiplication should be ∞. See Exercise 11.8 for more pitfalls in floating-point optimizations.

Modern processors invariably implement the floating point arithmetics specified in the IEEE 754 standard [128], and consequently modern language definitions support, or even mandate, IEEE 754 floating point semantics.

11.2.2 Enumeration types

An **enumeration type** defines a set of names to be the values of the new data type. The run-time representation is an integer, with values usually ranging from 0 to the number of enumeration values minus 1, although some languages allow the programmer to specify the integer values corresponding to the names of the enumeration type. In any case, the range of values is known at compile time, so an integer type of suitable size can be chosen to represent the values.

Operations allowed on enumerations usually are limited to copying, comparison for equality, comparison for greater/smaller, and sometimes increment/decrement, all of which are readily available in the target language for the integer representation chosen.

An enumeration type which is available in many languages, including some that do not have explicit enumeration types, is the **Boolean type**, with false and true as enumeration literals. An implementation with 0 for false and 1 for true suggests itself, but in many cases representing the Boolean value can be avoided, as shown in Section 11.4.1.1.

11.2.3 Pointer types

Most imperative and object-oriented languages support a **pointer type** of some kind. Pointers represent the addresses of source language data structures. The run-time representation of a pointer is an unsigned integer of a size large enough to represent an address. The integer operations for copying, assignment, comparison, etc. are available for pointers as well, and some target machines have special instructions or addressing modes for dealing with pointers.

The one operation that is particular to pointers is **dereferencing**, which consists of obtaining the value of the data structure that the pointer refers to. If the value to be obtained is small enough to fit in a register, dereferencing can usually be implemented as a single machine instruction. If the value is larger, though, as in dereferencing a pointer to a record or array, it is more efficient to find out what the final destination of the value is and to copy it to that place directly. For example, the assignment

```
q = *p;
```

in which q is the name of a record variable of type T and p is a pointer to a record of the same type, may be translated to a call of a library routine byte_copy():

```
byte_copy(&q, p, sizeof (T));
```

or the loop inside byte_copy may be in-lined.

An obvious optimization is available when the record obtained by dereferencing is used only for field selection. The language C has a special notation for this situation, ptr−>field, but other languages require the programmer to use the notation (*ptr).field. Literal implementation of the latter would indeed dereference the record under the pointer ptr to the top of the working stack and then replace the top of the working stack by the selected field. But when the compiler recognizes the situation it can first compute the pointer to the field and then dereference that pointer: (*ptr).field is translated as *(&(ptr−>field)).

The above applies when the context requires an rvalue. When the context requires an lvalue for a field selection, for instance in an assignment to ptr−>field, the required address can be obtained by adding the offset of field within the record to the pointer ptr.

Most languages that support pointers actually support **typed pointers**, for example POINTER TO INTEGER. In addition, there is often a **generic pointer type**, one that can be coerced to any other pointer type. For example, C has a "pointer to void" type, Modula-2 has a "pointer to byte" type. Coercing such a generic pointer type to another pointer type is a compile-time action, for which no run-time code is required.

11.2.3.1 Bad pointers

In the above we have assumed that a pointer indeed points to a valid value present in memory. In an incorrect program this need not be the case. Although the compiler can usually make sure that *if* the pointer refers to a value, that value will be of the right type, there are a number of situations in which the pointer does not refer to a value:

1. the pointer was never initialized;
2. the pointer is a null pointer, normally used to indicate the absence of a value, but the programmer forgot to check;
3. the pointer once referred to a value, but that value was located on the heap and has been removed since by a free() operation, leaving the pointer dangling;
4. the pointer once referred to a value, but that value was located in the activation record of a routine that has since been terminated, leaving the pointer dangling.

These dangers make pointers hard to use, as anyone with programming experience in C can confirm.

Languages differ in their approach to bad pointers. In C, the actions of an incorrect program are undefined, and the best one can hope of any of the above errors is that the program crashes before it produces incorrect results. The memory management units (MMUs) of most processors will generate an exception or trap if the pointer to be dereferenced is null, thus catching error 2 above, but their behavior on uninitialized pointers (error 1) will be erratic, and since errors 3 and 4 involve perfectly good pointers whose referents have gone away, no MMU assistance can be expected for them.

Language designers sometimes take measures to tame the pointer. Several approaches suggest themselves. Avoiding pointers at all is a good way to solve pointer problems (as done, for example, in functional and logic languages) but requires a lot of alternative programming support. In Java, pointers have been tamed by not allowing pointer arithmetic and limiting the way they can be obtained to object creation methods. Also, objects cannot be explicitly removed: as long as a pointer to an object exists, the object is alive. However, null-pointer dereferencing is still possible in Java.

Automatic initialization of pointers eliminates uninitialized pointers (error 1). The dereferencing of null pointers (error 2) can be eliminated in the language design by having two kinds of pointer-like type constructors, "pointers", which may point to a value or be null, and "references", which are guaranteed to point to a value. References can be dereferenced safely. Pointers cannot be dereferenced and can only be used in a test for null; if the pointer is not null, a reference is obtained, which can then, of course, be dereferenced safely. Having a garbage collector and disallowing calls to `free()` eliminates dangling pointers to the heap (error 3). And disallowing pointers to local variables eliminates other dangling pointers (error 4).

Although these possibilities are language design issues rather than compiler design issues, they are still relevant to compiler design. Implementing automatic initialization of pointers is easy, and symbolic interpretation can often establish that a pointer cannot be null. This is especially important in translating generated code, in which care has already been taken never to dereference null pointers. Also, there is a technique that avoids dangling pointer errors due to returning routines *without* disallowing pointers to local variables. Aspects of this technique are important for the run-time systems of some imperative and many functional and logic languages, which is why we will discuss these aspects here.

11.2.3.2 Pointer scope rules

Of the four types of errors, error 4 is the most problematic, since a good pointer turns into a dangling pointer through a routine exit, an action that is only remotely associated with the pointer. Following the saying that an ounce of prevention is worth a pound of cure, a set of rules has been developed to prevent dangling pointers from arising in the first place, the so-called "scope rules". Although no language except Algol 68 has put these rules in the hands of the programmer, they play a considerable role in the more advanced forms of routine construction and calling, and in the implementation of functional and logic languages. Because the possibilities and restrictions of passing routines as parameters and returning them as values are difficult to understand without these scope rules, it seems advisable to treat them here, in their more easily understandable data structure version.

Values located in activation records have limited "lifetimes", where the **lifetime** of an entity is the time in the run of the program during which the entity exists. The lifetime of a value is the same as that of the activation record in which it resides. Therefore, pointers to local values have limited "validity spans"; the **validity span** of a pointer is equal to the lifetime of the value it points to. For historic reasons, the validity span of a pointer is called its **scope**, and the rules for avoiding dangling pointers are called the **scope rules**. This terminology is unfortunate, since it inevitably causes confusion with the terms "scope of an identifier" and "scope rules", as discussed in Section 11.1.1, but we will conform to it to keep in line with the literature. Where necessary, a distinction can be made by using the terms "identifier scope" and "pointer scope".

The values that pointers refer to can be located in activation records or on the heap; for the purpose of this discussion, the global data area can be considered as the oldest activation record.

The lifetime of a value is equal to that of the container it is located in. For values on the heap, the lifetime is infinite; this does not mean that the data will be kept infinitely long, but rather that it is impossible to ever find out that its lifetime may not be infinite.

The lifetime of an activation record is governed by the routine calling mechanism; if the call is a simple subroutine call, the lifetime of the resulting activation record is enclosed in the lifetime of the activation record of the caller. If we call the lifetime of an activation record its "scope", as we did before with pointers, we see that one scope can lie completely inside another scope. We will call a scope P smaller than a scope Q ($P < Q$) if P lies entirely inside Q. This imposes an ordering on scopes, but we will see that some scopes, for example those in different threads, are incommensurable, so the ordering is a partial one in principle.

In summary, the lifetime of a value is the time span during which the value exists, the scope of a value is the time span during which the value is valid. The purpose of the scope rules is to make sure that any value will be valid during its entire lifetime.

We are now in a position to formulate the scope rules [299]:

- The scope of a location on the heap is infinite; the scope of a location in an activation record is that of the activation record.
- The scope of the activation record of the program is infinite; the scopes of the other activation records depend on the calling mechanism and the implementation.
- The scope of a value is the smallest scope of any pointer it contains or infinite if it does not contain pointers.
- The scope of a pointer is that of the location into which it points.
- A value with scope V may be stored only in a location with scope L if $V >= L$; in other words, `p := q` requires "scope of `p` <= scope of `q`".

It is incorrect to express the last rule as "A value with scope V may not be stored in a location with scope L if $V < L$". The assignment is also forbidden if V and L cannot be compared, since that would imply the possibility that part of the lifetime of L falls outside the lifetime of V.

Together these rules ensure that values remain valid as long as they exist, so no dangling pointers can originate from routines returning. If activation records are put on a stack, the scope of the activation record of a called routine is smaller than that of the activation record of the caller. An immediate consequence of that is that data on the heap cannot contain pointers to the stack, except perhaps to the global activation record.

Another consequence is that a local pointer variable in a routine *R* cannot point to data local to a routine called by *R*. This effectively prevents the notorious dangling pointer caused in C by assigning the address of a local variable to a pointer passed as a parameter, as shown in Figure 11.14. Here the scope of &actual_buffer is that of the activation record of obtain_buffer(), and that of *buffer_pointer is that of do_buffer(), which is larger. This assignment constitutes a scope rule violation; if it is allowed to pass unchecked, as it is in C, the following happens. When obtain_buffer() returns, actual_buffer[] disappears, but a pointer to it remains in buffer, which is passed to use_buffer(). When use_buffer() then uses its now invalid parameter, the scope rule violation takes its toll.

```
void do_buffer(void) {
    char *buffer;

    obtain_buffer(&buffer);
    use_buffer(buffer);
}

void obtain_buffer(char **buffer_pointer) {
    char actual_buffer[256];

    *buffer_pointer = &actual_buffer;
        /* this is a scope-violating assignment: */
        /* scope of *buffer_pointer > scope of &actual_buffer */
}
```

Fig. 11.14: Example of a scope violation in C

In principle the scopes of pointers, locations, and values could be maintained and checked at run time, but doing so is awkward and inefficient. If the need arises, as it does in Algol 68, symbolic interpretation can be used to avoid generating checks where they are not needed. In practice the scope rules are used at compiler design time, to design efficient implementations of advanced routine operations in all paradigms with the property that no dangling pointers will ever be generated. These implementations are designed so that they never store a short-lived value in a longer-lived container. We will see several examples in Section 11.3.

A run-time approach to pointer safety is taken by Austin *et al.* [21], who discuss a method for the detection of all pointer and array access errors by replacing all pointers by so called “safe pointers”. Oiwa [207] describes the extensive effort that went into the design and construction of a memory-safe full ANSI-C compiler, *Fail-Safe C*. Programs compiled with this compiler are on the average 5 times slower than with *gcc*.

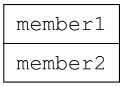
11.2.4 Record types

A **record**, also called a **structure**, is a data item in which a fixed number of members, also called “components”, of possibly different types are grouped together. In the target language, these members are represented consecutively in a memory area that is large enough to contain all members. What constitutes “large enough” depends on the sizes of the members and the alignment requirements of the target machine.

For example, a variable of the C structure

```
struct example {
    int member1;
    double member2;
};
```

is—in principle—represented as follows:

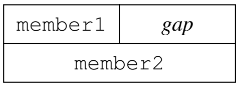


As explained in Section 8.1.4, some processors impose address alignment requirements on certain data accesses. For example, the SPARC processor requires an int (a 4-byte quantity) to have an address that is a multiple of 4 (is 4-byte aligned), and a double (an 8-byte quantity) to be 8-byte aligned. member1 of the example above thus must be 4-byte aligned, which is easily accomplished if the structure itself is 4-byte aligned, but member2 must be 8-byte aligned, and this can be accomplished by first making sure that it is 8-byte aligned within the structure by inserting a gap of 4 bytes between member1 and member2, and then making sure that the structure itself is 8-byte aligned. This way, member2 will also always be 8-byte aligned.

In general, gaps must be inserted between the structure members to make sure that each member is aligned properly within the structure, and then the size and the alignment requirement for the structure itself must be computed. This alignment requirement is the lowest common multiple (LCM) of the member alignment requirements. Often this is just the largest member alignment requirement, because alignment requirements usually are small powers of 2 (and thus one is either a multiple or a divisor of the other). Also, a gap can be inserted at the end of the structure to make sure that the size of the structure is a multiple of its alignment requirement.

This is convenient when the structure is used to build other data types, such as an array of these structures.

To get back to our example structure, on a SPARC it will be represented as follows:



and it must be aligned on 8-byte boundaries. If x is a variable of this type, it has size 16, x.member1 lies at offset 0 from the start of x, and x.member2 lies at offset 8 from the start of x.

The mandatory operations on record types are **field selection** and copying; comparison for equality is not always required. The selection of a field from a record is accomplished in the target language by computing the address of the field, and then accessing the field through that address. This is done by adding the offset of the field within the record, which is known to the compiler, to the address of the record. The resulting address is the lvalue of the field (if it has one in the context at hand). To obtain the rvalue of the field, the address must be dereferenced. Record copying can be accomplished by either copying all fields individually or by copying the whole record, including the contents of any gaps. The latter may be more efficient if there is an efficient memory block copy instruction or routine available in the target language.

The possible presence of gaps makes record comparison more complicated than record copying. The contents of a gap usually are undefined, so they must be disregarded when comparing records. Therefore, record comparison has to be done field by field. For the example record type above, the compiler could generate a routine, returning 1 if the records are equal, and 0 if they are not, according to the following C scheme:

```
int compare_example(struct example *s1, struct example *s2) {
    if (s1−>member1 != s2−>member1) return 0;
    if (s1−>member2 != s2−>member2) return 0;
    return 1;
}
```

and then generate code that calls this routine whenever two records of this type must be compared. It could also generate in-line code to perform the comparison.

11.2.5 Union types

A **union type** is a data type of which the values are of one of a set of types. For example, a variable a of type

```
union {
    int i;
```

```
        float  f;
    };
```

in C can either hold a value of type int or a value of type float. To access the value of type int, the programmer uses a.i; to access the value of type float the programmer uses a.f. A union cannot hold both values at the same time. In C, the programmer is responsible for keeping track of which of the union fields is present: the union is not "discriminated". Some other languages have a special field, called the **union tag**, which is always present in unions and indicates which variant the union currently holds: the union is "discriminated". Each variant of the union has its own (constant) value of the union tag. The type of the union tag usually is an enumeration type, and each variant is associated with a particular enumeration value.

The run-time representation of an undiscriminated union is very simple: it is like a record, except that all fields overlap. Therefore, its size is equal to the aligned size of the largest variant, and its alignment requirement is equal to the lowest common multiple of the alignment requirements of its variants. Field selection of an undiscriminated union is simple: the field always resides at offset 0 from the start of the union. Copying consists of a memory copy of the size of the union. Comparison is not available for undiscriminated unions.

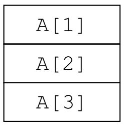
The representation of a discriminated union is that of a record containing two fields: the union tag and the—undiscriminated—union of the variants, as above. The generated code can access the union tag to check which variant is current; whether the programmer can also do so is source-language dependent. In principle a check must be generated for each union-field access to make sure that the selector matches the current variant of the union, and if it is not, a run-time exception must be produced. In an optimizing compiler this check can often be avoided by doing static analysis, for example using symbolic interpretation as explained in Section 5.2.

11.2.6 Array types

An **array type** describes data structures which consist of series of items (also called **elements**) of the same type. An array can have one or more dimensions. An array element is indicated through one or more index expressions, one for each dimension of the array. The run-time representation of the array consists of a consecutive sequence of the representation of the elements; the address of the first element is called the **base address** of the array. For example, a one-dimensional array

```
A: ARRAY [1..3] OF Integer;
```

will be stored as



and a two-dimensional array

```
B: ARRAY [1..2, 1..3] OF Integer;
```

can be stored in either **row-major order**, which means that the elements are stored row after row (Figure 11.15(a)), or in **column-major order**, which means that the elements are stored column after column (Figure 11.15(b)). These schemes can easily be extended to more than two dimensions. Below, we will assume row-major order, as this is the order most often used.

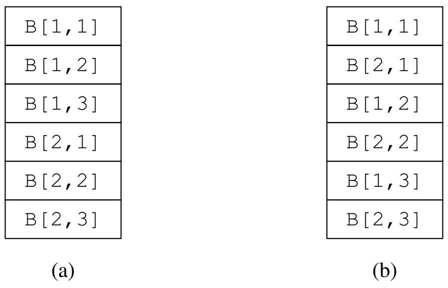


Fig. 11.15: Array B in row-major (a) and column-major (b) order

Note that, as long as the compiler makes sure that the element size is a multiple of the alignment requirement of the element, there will be no gaps *between* the array elements. This is important for the implementation of array comparison, as well as the implementation of element selection, as discussed below.

As with records, array copying can be done either element by element or through a memory block copy. Array comparison, if the source language allows it, can be done through a direct memory comparison if the array elements do not have any gaps, and element by element otherwise.

All languages that support arrays also support **element selection**, which is usually called **indexing**. Assume an n-dimensional array A with base address $base(A)$, where dimension k has lower bound LB_k and upper bound UB_k. The number of elements along dimension k is then $LEN_k = UB_k - LB_k + 1$. Now suppose that the location of element $A[i_1, i_2, \ldots, i_n]$ must be computed. Given the base address $base(A)$ and the size *el_size* of the array elements in bytes, the location of the required element is obtained by multiplying *el_size* with the number of elements in front of the required element, and adding the result to $base(A)$. For a one-dimensional array this location is $base(A) + (i_1 - LB_1) \times el\_size$. In general, for an n-dimensional array, the location of the element $A[i_1, i_2, \ldots, i_n]$ is:

$$
\begin{aligned}
& base(A) + \quad ((i_1 - LB_1) \times LEN_2 \times LEN_3 \times \ldots \times LEN_n \\
& + (i_2 - LB_2) \times LEN_3 \times \ldots \times LEN_n \\
& \ldots \\
& + (i_n - LB_n) \ ) \times el\_size
\end{aligned}
$$

This is a lot of computation just to determine the location of one element. Fortunately, a large part of it can be precomputed. Reorganizing the expression results in

$$\begin{aligned}
&base(A) - \ (LB_1 \times LEN_2 \times LEN_3 \times \ldots \times LEN_n \\
&+ LB_2 \times LEN_3 \times \ldots \times LEN_n \\
&+ \ldots \\
&+ LB_n \ ) \times el\_size \\
&\ + \ (i_1 \times LEN_2 \times LEN_3 \times \ldots \times LEN_n + i_2 \times LEN_3 \times \ldots \times LEN_n + \ \ldots + \ i_n\,) \times el\_size
\end{aligned}$$

All lines of this expression except the last are independent of the indices in the element selection and depend only on the array A. In fact, the first n lines of the expression contain the location of the element $A[0,0,\ldots,0]$, the **zeroth element**.

Of course, it is quite possible that the zeroth element does not exist inside the array, because 0 may not be a member of the interval $LB_k \ldots UB_k$ for some k; this does not prevent us, however, from using its location, *zeroth_element*, in address computations. The $LEN_k \times \ldots \times LEN_n$ products in the last line are also independent of the indices, and we can compute them in advance. If we designate the product $LEN_k \times \ldots \times LEN_n$ by $LEN\_PRODUCT_k$, we have:

$$\begin{aligned}
&LEN\_PRODUCT_n \ = \ el\_size \\
&LEN\_PRODUCT_k \ = \ LEN\_PRODUCT_{k+1} \times LEN_{k+1} \qquad \text{for } n > k >= 1 \\
&zeroth\_element(A) \ = \ base(A) - \ (LB_1 \times LEN\_PRODUCT_1 \\
&+ LB_2 \times LEN\_PRODUCT_2 \\
&+ \ldots \\
&+ LB_n \times LEN\_PRODUCT_n \ )
\end{aligned}$$

With these values precomputed, the location of the element $A[i_1,i_2,\ldots,i_n]$ can be computed by the formula

$$zeroth\_element(A) \quad + i_1 \times LEN\_PRODUCT_1 \quad + \ \ldots \quad + i_n \times LEN\_PRODUCT_n$$

All these precomputed values for the array can be stored in an **array descriptor**. This descriptor should also contain the array bounds (or the lower bound and the length) themselves, so array bound checks can be generated. An example of an array descriptor is shown in Figure 11.16.

The computation described above results in the location of an array element. When an lvalue is required in the context at hand, this location serves as one. When the rvalue of the array element is required, the location (a pointer) must be dereferenced. Again, this dereferencing may be implicit in the selected machine instructions.

When the array does not change size during its lifetime, and the bounds are known at compile time, it is called a **static array**. In contrast, when the array may change size during its lifetime, or its bounds are determined at run time, it is called

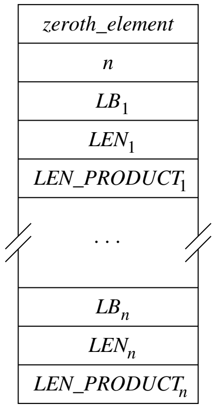


Fig. 11.16: An array descriptor

a **dynamic array**. For static array access the array descriptor does not actually have to be used at run time. Instead, the compiler can compute all the values the array descriptor would contain and use them as constants in index expressions, with the exception of *zeroth_element*. The compiler *can*, however, compute the offset of *zeroth_element* from the first element of the array itself, and use that to compute the value of *zeroth_element* when needed.

11.2.7 Set types

Some languages have **set types**, which are usually limited to sets of a small subrange of integers. These are probably best implemented as bitsets. In a **bitset**, each value in the subrange is represented by a single bit: if the bit is set, the value is a member of the set; if the bit is not set, the value is not a member of the set. Bitsets can be stored conveniently in target machine words. For example, the set of the integers ranging from 0 to 31 can be represented by a single 32-bit word.

The usual set operations are implemented by means of bit operations: set union is implemented with a bitwise OR, set intersection is implemented with a bitwise AND, symmetric set difference is implemented with a bitwise EXCLUSIVE OR, and set difference is implemented with a bitwise NOT followed by a bitwise AND.

Some languages allow sets of any type for which comparison for equality exists. Some possible representations of such sets are linked lists [275], trees, and hash tables [251].

11.2.8 Routine types

Routines are not considered data in many programming languages, and in these languages the question of how to represent routines as a data type does not arise. In C, a routine as data can be implemented simply as a pointer to its code; an indirect routine call is then used to activate it. The best implementation of routines as data in languages that allow more advanced operations on routines—nesting, routines as first-class data, partial parameterization, etc.—depends in complicated ways on the exact operations allowed and the choices made in the implementation as to the allocation of the activation records. The issue will be covered integrally in Section 11.3.

11.2.9 Object types

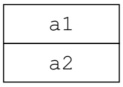
An **object** is a record with built-in methods, with some additional features. Its type is usually called a **class**. In some languages, an object is similar to an *abstract data type* in which only the methods are visible from outside and the object fields can only be accessed from within the method bodies. In other languages, the object fields can be accessed like record fields.

The basic operations on objects are "field selection", "copying", and "method invocation". Although many object-oriented languages do not allow direct access to the fields of an object, field selection is needed for the access from inside methods in the object. Copying is trivial and will not concern us here. For **method invocation**, first the method must be identified, and then the call must be made.

The method call is similar to a routine call, but to make the object fields directly accessible from within the method body, a pointer to the object is passed implicitly as an additional parameter. Within the method the object is usually visible under a reserved name such as self or this; the method can then access the object fields through this name. In this section we will focus on method selection; the actual routine or method invocation is discussed in Section 11.4.2.

In most object-oriented languages, objects also have **constructors** and **destructors**, which are methods that are to be invoked on object creation and object removal, respectively. As far as the following discussion is concerned, these are methods which are invoked like any other method.

If this was all there was to it, implementation of objects would be quite simple. Suppose that we have an object class A with methods m1 and m2 and fields a1 and a2. The run-time representation of an object of class A then consists of a record containing the fields a1 and a2:



In addition, the compiler maintains a compile-time table of methods for class A:

m1_A
m2_A

where we have appended "_A" to the method names, to indicate that they operate on an object of class A, rather than on a different class that also defined methods with names m1 or m2. In this simple model, field selection is implemented as record field selection, and object copying is implemented as record copying; method selection is done by the identification phase in the compiler. Methods are implemented as routines with one additional parameter, a pointer to the object. So the method m2_A could be translated to the C routine

```
void m2_A(Class_A *this, int i) {
    Body of method m2_A, accessing any object field x as this−>x
}
```

assuming that m2_A() has one integer parameter and returns no value and where Class_A is the C type name for class A. The method invocation a.m2(3); is then translated to m2_A(&a, 3);.

The problem is, however, that all object-oriented languages have at least some of the features discussed below. They make objects much more useful to the programmer, but also complicate the task of the compiler writer. These are the features that distinguish objects from abstract data types.

11.2.9.1 Feature 1: Inheritance

Inheritance, present in all object-oriented languages, allows the programmer to base a class B on a class A, so that B inherits the methods and fields of A, in addition to its own fields and methods. This feature is also known as **type extension**: class B extends class A with zero or more fields and methods. Class A is the **parent class** of class B, and class B is a **subclass** of class A. Now suppose that class B extends class A by adding a method m3 and a field b1. The run-time representation of an object of class B is then:

a1
a2
b1

In addition, the compile-time table of methods for class B is:

m1_A
m2_A
m3_B

This can still be implemented using the means for abstract data types described above.

11.2.9.2 Feature 2: Method overriding

When a class B extends a class A, it may redefine one or more of A's methods; this feature is called **method overriding**. Method overriding implies that when a parent class *P* defines a method, all classes based directly or indirectly on *P* will have that method, but the implementations of these methods in these subclasses may differ when overridden by redefinitions. Put more precisely, the method is *declared* in the class *P*, and then *defined* in class *P* and possibly redefined in any of its subclasses. We use the phrase "the declaration of *X*" here as a statement saying that *X* exists, and "the definition of *X*" as a statement telling exactly what *X* is; this usage is in accordance with but somewhat stricter than the traditional usage in programming languages, where one speaks for example of "forward declarations" rather than of "forward definitions".

Now assume that class B in the example above redefines method m2, which was already defined for objects of class A. Then the definition of method m2 in A is both its only declaration and its first definition; that in class B is a redefinition. Some languages, for example C++ and Java, allow the method declaration to occur without a definition; the method is then a **virtual** or a **abstract method** (terminology depending on the language at hand), and a class in which at least one virtual method occurs is an **abstract class**. The actual methods must then be defined in classes that extend the abstract class.

We will rename the methods so that the name reflects both the class in which it is declared and the class in which it is defined. The names consist of three parts, the method name, the class it is declared in, and the class it is defined in; the parts are separated by underscores (_). So a name m2_A_B is used to designate a method m2 declared in class A and defined in class B.

Method overriding affects the compile-time table of methods. Under the above assumption that class B redefines method m2, which was already declared and defined in class A, the method table of class A now becomes:

m1_A_A
m2_A_A

and the method table of class B becomes:

m1_A_A
m2_A_B
m3_B_B

Now suppose a is an object of class A, and b is an object of class B. A method call a.m2(...) will then be translated with a call to m2_A_A, whereas a method call b.m2(...) will be translated to a call to m2_A_B. This differentiates clearly between m2_A_A, which was declared in class A and defined in class A, and m2_A_B which was also declared in class A but defined in class B.

If inheritance is the only other object-oriented feature in the language, the type of the translation of m2_A_A is

```
void m2_A_A(Class_A *this, int i);
```

and that of m2_A_B is

```
void m2_A_B(Class_B *this, int i);
```

11.2.9.3 Feature 3: Polymorphism

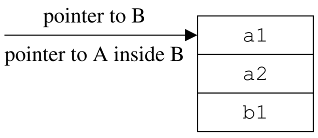
When a class B extends a class A and the language allows a pointer of type "pointer to class B" to be assigned to a variable of type "pointer to class A", the language supports **polymorphism**: a variable of type "pointer to class A" may actually refer to an object of class A or any of its extensions. The implementation of this feature requires a new operation, **pointer supertyping**: converting a pointer to an object of subclass B to a pointer to an object of a parent class A. This operation is used in assignments, for example:

```
class B *b = ...;
class A *a = b;
```

in which the second line is translated into

```
class A *a = convert_ptr_to_B_to_ptr_to_A(b);
```

For now, the routine convert_ptr_to_B_to_ptr_to_A() is a compile-time type operation. Because an object of class B starts with the fields of class A, the value of the pointer need not be changed and the only effect consists of changing the type of the pointer:



But note that we now have identical pointers to objects of different classes.

11.2.9.4 Feature 4: Dynamic binding

Now that a pointer p of type class A * can actually refer to an object of class B, a question arises about the methods that apply. If the source program applies method m2 to the object referred to by p, should the translation invoke m2_A_A or m2_A_B? There are two possible answers to this question: **static binding**, which maintains that statically p refers to an object of class A, so m2_A_A should be called; and **dynamic binding**, which maintains that if the object is actually of class B we should apply m2_A_B, and if the object is actually of class A, we should apply m2_A_A.

Static binding is trivial to implement, but most object-oriented languages use dynamic binding, for various important programming language design reasons. Dynamic binding has two significant consequences:

- There are two kinds of As out there, "genuine" As which use m2_A_A and As "embedded" in Bs, which use m2_A_B, and the two cannot be distinguished statically. It follows that from now on, the object representation must include dynamic type information, telling if it is an A or a B.
- B's methods require a pointer to a B to obtain access to all of B's fields. As m2 may, however, be called through a pointer of type "pointer to A" that dynamically points to an object of class B, we need yet another operation, **pointer (re)subtyping**, which reconstructs a pointer to B from the pointer to A. The method invocation p−>m2(3), where p is statically a pointer to an object of class A, could then be translated to

  ```
  switch (dynamic_type_of(p)) {
      case Dynamic_class_A: m2_A_A(p, 3); break;
      case Dynamic_class_B:
          m2_A_B(convert_ptr_to_A_to_ptr_to_B(p), 3); break;
  }
  ```

  where the dynamic type information is an enumeration type with the values Dynamic_Class_A and Dynamic_Class_B. When p is statically a pointer to B, we could translate the invocation p−>m2(3) immediately to

  ```
  m2_A_B(p, 3);
  ```

  Note that this code is consistent with the declarations void m2_A_A(Class_A *this, int i) and void m2_A_B(Class_B *this, int i). We will see, however, that a better translation is possible. For now, pointer subtyping is again a compile-time operation.

The switch statement used to find out which method routine to call is a function that works on a small domain, and lends itself to precomputation. To this end we incorporate the pointer conversion from A to B in the routine for m2_A_B, which now accepts a pointer to A:

```
void m2_A_B(Class_A *this_A, int i) {
    Class_B *this = convert_ptr_to_A_to_ptr_to_B(this_A);
    Body of method m2_A_B, accessing any object field x as this−>x
}
```

More in general, every method translation m_X_Y gets as its first parameter a pointer to Class_X, which is then immediately converted to a pointer to Class_Y by applying convert_ptr_to_X_to_ptr_to_Y(). If X and Y are the same, the conversion can be omitted.

With this modification to m2_A_B(), the method invocation p−>m2(3), where p is statically a pointer to an object of class A, can be translated as

```
(dynamic_type_of(p) == Dynamic_class_A ? m2_A_A : m2_A_B)(p, 3);
```

which features a computed function which is called with the parameter list (p, 3). Rather than computing the function from the dynamic type information of p each time an operation on p is performed, we can incorporate the resulting function address in the dynamic type information. The type information for an object of class B is then a record with three selectors, m1_A, m2_A, and m3_B, containing the addresses of the routines to be called for methods m1, m2, and m3. These are m1_A_A(), m2_A_B(), and m3_B_B(); each of these routines has as its first parameter a pointer to an object of class A. Such a record with addresses of method routines is called a **dispatch table** and the type information in each object is implemented as a pointer to its dispatch table, as shown in Figure 11.17.

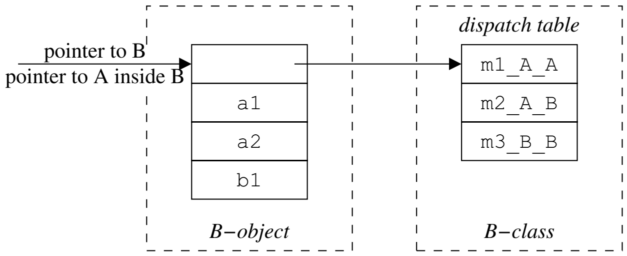


Fig. 11.17: The representation of an object of class B

The type information for an object of class A is a two-field dispatch table containing the addresses of the routines m1_A_A() and m2_A_A(), both of which have a pointer to an object of class A as their first parameter. So in both dispatch tables the selector m1_A selects routines of the same type; the same applies to m2_A. This in turn means that the selection p−>dispatch_table−>m2_A, where p is of type class A *, yields the proper routine with the proper type, regardless of whether p points to an object of class A or one of class B. The method invocation p−>m2(3) can now be translated very efficiently as

```
(p−>dispatch_table−>m2_A)(p, 3);
```

As we have modified the routine m2_A_B(), the translation given above for the invocation p−>m2(3), when p is statically a pointer to B, is no longer correct. The routine now expects a pointer to A, and must be given one:

```
m2_A_B(convert_ptr_to_B_to_ptr_to_A(p), 3);
```

Until now, the effect of the conversion routines has been that of type conversion only; no actual code was involved. We will now turn to another object-oriented feature that requires the conversion routines to have substance.

11.2.9.5 Feature 5: Multiple inheritance

So far, we have only discussed **single inheritance**, in which an object class may only inherit from a single parent class. In this section, we will discuss the consequences of allowing objects to extend more than one parent class. This feature is called **multiple inheritance**, and is supported by several important object-oriented programming languages. Suppose, for example, that we have an object class C with fields c1 and c2 and methods m1 and m2, an object class D with field d1 and methods m3 and m4. Next an object class E is defined which extends both C and D, adds a field e1, redefines methods m2 and m4, and adds a method m5. All this is shown in Figure 11.18.

```
class C {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};
```

Fig. 11.18: An example of multiple inheritance

Unlike the situation with single inheritance, it is no longer possible to represent an object as a pointer to a dispatch table followed by all object fields. In particular, the "D inside E" object must start with a pointer to its dispatch table, followed by its object fields. It is still possible to combine the dispatch tables for E, "C inside E", and "D inside E". However, they are no longer all indexed by the same pointer. The dispatch table for E becomes:

```
m1_C_C
m2_C_E
m3_D_D
m4_D_E
m5_E_E
```

so the dispatch tables for E and "C inside E" still have the same address, but the pointer to the dispatch table for "D inside E" refers to the third entry (that of m3_D_D of the dispatch table for E. Moreover, this pointer has to be represented explicitly between the object fields of E. The representation is summarized in Figure 11.19.

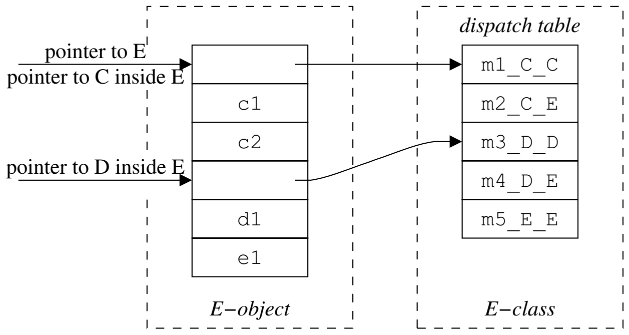


Fig. 11.19: A representation of an object of class E

This has consequences when viewing an object of class E as an object of class D, and vice versa. The pointer supertyping and pointer subtyping operations, which up to this point did not require code, now in some cases get substance:

```
supertyping:
    convert_ptr_to_E_to_ptr_to_C(e) ≡ e
    convert_ptr_to_E_to_ptr_to_D(e) ≡ e + sizeof (class C)

subtyping:
    convert_ptr_to_C_to_ptr_to_E(c) ≡ c
    convert_ptr_to_D_to_ptr_to_E(d) ≡ d − sizeof (class C)
```

When an object class E inherits from both class C and class D, an ambiguity may arise. For example, C and D may both contain a method with the same name. Class E inherits both these methods, but when applying a method with this name to an object of class E, only one method can be applied, and it may not be clear which version is intended. The language rules should indicate when a conflict or ambiguity arises, and the compiler has to detect these situations. Often the compiler task is complicated enough to warrant specific publications, for example Ramalingam and Srinivasan [230] or Boyland and Castagna [48]. These are very language-specific problems, however, and we will not discuss them here.

11.2.9.6 Feature 6: Dependent multiple inheritance

An important issue that arises with multiple inheritance, one which we have ignored in the discussion above, is **repeated inheritance**. For example, if both class C and D of the example above are extensions of a class A, as depicted in Figure 11.20, then what does "A inside E" mean? Depending on the language, this can mean one of two

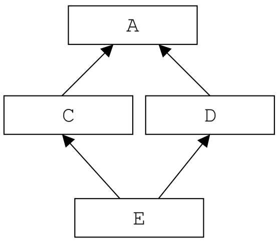


Fig. 11.20: An example of repeated inheritance

things: either an object of class E contains two objects of class A, or it contains one object of class A. In the former case we speak of **independent inheritance**, in the latter we speak of **dependent inheritance**. Some languages allow both, or even a mixture of the two: some fields may be inherited dependently, others independently.

Independent multiple inheritance is implemented exactly as described above. The only complication is in the identification, and the language rules should prescribe when and how the programmer should use qualification, exactly as if the fields and/or methods of class A resulted from different classes, but with the same names.

Now let us turn to dependent inheritance. The added complexity here is not in the method selection, but in the representation of the object data. We can no longer use the "parent objects first" scheme, because, in the example above, we would get two copies of the data of A. So, we will have to arrange the object data so that it only has one copy of the A data. We can accomplish that by placing the components of the object data in the object in the following order: the first entry is the pointer to dispatch table of E (and of "C inside E"); the next entries are the fields of A; then follow all fields of C not inherited from A; the entry after that is the pointer to dispatch table of "D inside E" (which points to within E's dispatch table); next are all fields of D not inherited from A; and, finally, all fields of E not inherited from C or D.

This order is correct for the E object and the "C inside E" object, but what about the "D inside E" object? The compiler has to decide what a D object looks like when compiling the D object class, at a time when it does not know about either C or E. Assuming that a D consists of a pointer to its dispatch table followed by its fields will not work, since when a D resides inside an object of type E, the fields it inherits from A are some distance ahead of the dispatch table pointer and D's own fields follow it. So when producing code to access the object fields, the compiler has no idea where they are. As usual, the answer to this kind of problem is a run-time descriptor. This descriptor should allow a method to find the object fields, given a pointer to the object itself. For each field, the descriptor must contain the offset of the field from the object pointer. We enumerate the object fields, so we can use the enumeration index as an index in an **offset table**. In addition to a pointer to the dispatch table, the object representation now must also contain a pointer to the offset table. Since we do not know beforehand which object classes will be involved in multiple inheritance, we must follow this two-pointer scheme for all objects.

Now let us return to our example of Figure 11.18, and extend it by assuming that both object class C and object class D extend an object class A, which has fields a1 and a2 and methods m1 and m3. So, object class C redefines method m1, and object class D redefines method m3; see Figure 11.21.

```
class A {
    field a1;
    field a2;
    method m1();
    method m3();
};

class C extends A {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D extends A {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};
```

Fig. 11.21: An example of dependent multiple inheritance

An object of class E has the representation depicted in Figure 11.22. Assuming that all pointers and fields have size 1, field a1 has offset 2 from the E pointer, field a2 has offset 3, etc., and the offset table of class E contains

```
2 3 4 5 8 9
```

The offset table of the “D inside E” class contains

```
–4 –3 2
```

Note that for an object of class E, there is an ambiguity on m1 as well as m3. The language rules or the programmer will have to specify which m1 and m3 are intended when applying it to an object of class E.

By now it may be clear why some languages (notably Java) do not allow multiple inheritance: it causes many complications and adds some method invocation overhead. On the other hand, the benefit is added flexibility.

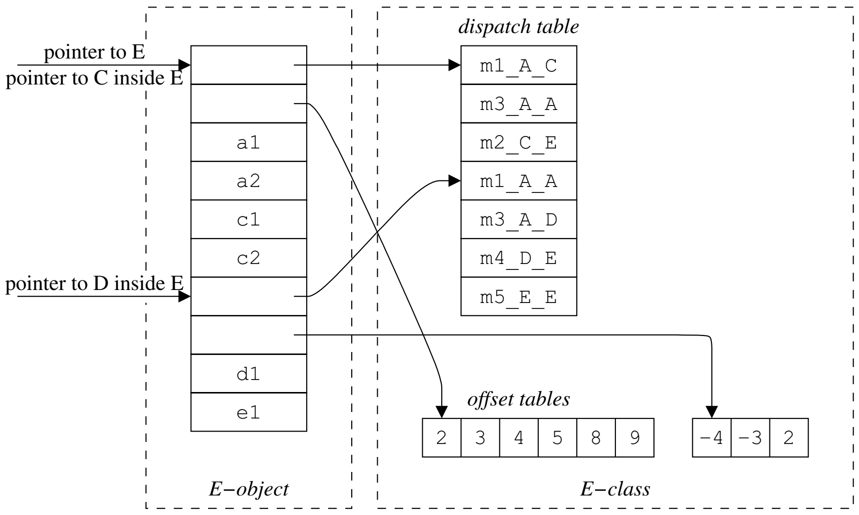


Fig. 11.22: An object of class E, with dependent inheritance

11.2.9.7 Optimizations for method invocation

All the manipulations discussed above make a method invocation more expensive than a routine call: the method is called through a dispatch table, and the method itself may have to adjust the object pointer that is passed to it. Therefore, some languages have a mechanism for indicating that a method may not be redefined by a subclass. Such an indication allows the compiler to identify the method directly, as it would identify any routine; the compiler can then use the routine call mechanism instead of the dispatch table. In Java, a method can be marked "'final'", indicating that it may not be redefined. In C++, methods that may be redefined must be marked "virtual".

Static analysis can also be of assistance in allowing the compiler to find out exactly to which subclass a method is applied. It may also be able to determine the method called, and generate a direct routine call.

More information about the implementation of inheritance can be found in papers by Templ [276], which covers independent multiple inheritance only, and Vitek and Horspool [289], which covers implementation of dynamic binding in dynamically typed object-oriented languages.

11.2.10 Interface types

Java has incorporated an extension that relieves the limitations of single inheritance somewhat, without adding the complexity (and all the power) of multiple inheritance. The extension consists of so-called interfaces. An **interface** is like an object class in that it consists of a number of method specifications. In contrast to an object class, however, it cannot have non-constant object fields and all methods must be abstract. An interface may extend a single parent interface.

An interface is not instantiated like an object, but one can declare Java variables of an interface type, and invoke methods from the interface specification on them; Java variables are actually pointers to objects. The trick is that an object class may specify that it implements one or more of those interfaces, and that an interface type is compatible with any object type that implements this interface. So, for example, given an interface

```
public interface Comparable {
    public int compare(Comparable o);
}
```

it is possible to define an object class that implements this interface, while it still can extend another object class.

The compiler must generate a separate dispatch table for each interface that an object class implements. This separate interface dispatch table only contains entries for methods that are specified in the interface specification, but the entries refer to methods of the object type. A variable of an interface type can be represented by a record containing two pointers, one to the interface dispatch table, and one to the object. Method invocation on an interface then goes through the pointer to the interface dispatch table. Conversion from an interface value to an object class value requires a run-time check to ensure that the object class actually corresponds to the type of the object referred to by the interface value, or is a parent class of it. The reverse conversion, from a class to an interface, consists of a compile-time check that the object class actually implements the interface type.

11.3 Routines and their activation

Routines have been with us since the first programs were written and will probably continue to serve us for a very long time. It is therefore amazing to see how complicated these seemingly basic entities are. A routine call is a combination, a successful combination one must admit, of at least four loosely related features:

1. supplying a new computing environment containing at least some temporary memory, the local variables;
2. passing information to the new environment, the parameters;
3. transfer of the flow of control to the new environment, with—in normal circumstances–an eventual return to the caller;
4. returning information from the new environment, the return value(s).

Some of these features are available in isolation in some languages. Creating a new environment (1) is available in C and many other languages as block entrance. Code that can be transferred to with guaranteed return (3) without creating a new environment is known as a “refinement” [26]. But the package deal of the routine call has been far more important.

Roadmap

11.3.1 Activation records

The list of the four features above shows that the basic ingredient of a routine activation is the new environment. The data structure supporting the new environment is the **activation record**, also called **frame**. An activation record holds the data pertinent to an invocation of a routine or object method; it represents an activated and not yet terminated routine. In particular, it contains user data—local variables, parameters, return values, register contents—and administration data—code addresses, pointers to other activation records. Managing the user data is discussed in Section 11.4.2.2.

In non-concurrent code, only one of the activation records represents a running routine; all the others are suspended: no instructions of those routine activations are being executed. The instruction being executed is located in the code of the running routine and the **program counter**, PC, points at it, or—more usually—just after it, depending on hardware conventions. The activation record of the running routine is indicated by a **frame pointer**, FP. The frame pointer, which usually resides in a dedicated register, is used at run time to access the contents of the activation record of the running routine; its use is explained in more detail in Section 11.4.2.2.

Depending on the language and function that is being compiled, the offset from the stack pointer to the frame pointer may be known at compile time. In such cases it may be more efficient to not maintain a frame pointer, but use the stack pointer instead.

In this section we concern ourselves with allocating, deallocating, and otherwise organizing the activation records in efficient ways. In many languages, routines are activated in a strictly last-in-first-out order: when a routine A invokes a routine B, A cannot continue until B has finished. For such languages a stack is the preferred allocation scheme for activation records. In other languages complications exist, due to features like nested routines, iterators, coroutines, routines passed as parameters, routines returned as routine results, partially parameterized calls, non-local gotos, and continuations. Such features complicate the use of a stack for the activation records. In our implementations, we will try to use stack allocation as long as possible, for efficiency reasons.

We will first discuss the contents of activation records. Next, we consider several forms of routines and make an inventory of the operations available on routines. Finally we consider implementations of these operations on the various forms of routines.

11.3.2 The contents of an activation record

Although the contents of an activation record depend on the source language, the compiler, and the target machine, the dependency is not a strong one, and all types of activation records have much in common. They almost invariably include the following components:

- the local variables of the routine;
- the parameters of the routine;
- the working stack; and
- an administration part.

The first three components correspond closely to the local variables, parameters, and intermediate results in the source program. These components will be examined in more detail in Section 11.4.2.2, where we discuss the actual routine call. Some aspects of the administration part are discussed further on in this subsection.

Figure 11.23 shows a possible structure for an activation record. The peculiar order of the components is helpful in the creation of activation records in routine calls and in addressing the user values they contain. It is the most natural order on most machines and it is the order used in our treatment of routine calls in Section 11.4.2.2. Other orders are possible, though, and may have their advantages, especially when they are supported by special stack manipulation instructions.

An activation record is accessed through its frame pointer. In our treatment this frame pointer points to the last byte of the administration part, just before the first local variable. This is convenient since it allows addressing local variable k by FP + offset(k) and parameter p by FP + sizeof (administration part) + offset(p), regardless of how many parameters or local variables there are. The offsets of local variables are negative, those of the parameters positive. The direction of low to high addresses is in accordance with that on most machines.

The exact contents of the administration part of an activation record *A* resulting from the invocation of a routine *R* are machine- and implementation-dependent. They always include either return or continuation information, and a "dynamic link"; they may include a "lexical pointer". In some implementations, the administration part also contains copies of values kept in registers, but since these again

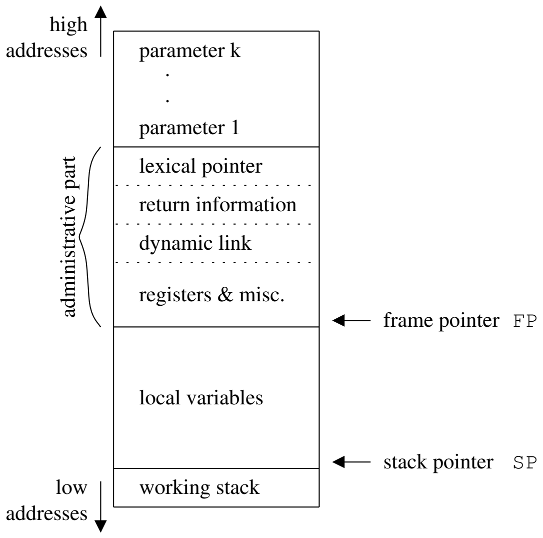


Fig. 11.23: Possible structure of an activation record

represent local variables, parameters, and intermediate results from the source program, their treatment is also deferred to Section 11.4.2.2. **Return information** consists of a return address, which is the code address in the caller of *R* to which the flow of control will return when *R* terminates. **Continuation information** consists of a continuation address, which is the code address in *R* at which the flow of control will continue when *R* is resumed. Which of the two is present depends on the implementation, as explained in the next few paragraphs. In both cases the **dynamic link** is the frame pointer of the caller of *R*.

The issue of storing return addresses in the activation records of callees or continuation addresses in the activation record of callers may be confusing and a few words are in order. In principle, any operation that suspends a routine *R* for whatever reason can store the continuation (resumption) information in the activation record of *R*; the information can then serve to continue *R* when that is required. Such operations include calling another routine, suspending in an iterator, transferring to another coroutine, and even a thread switch (Section 14.1.2). Storing continuation information about *R* in the activation record of *R* has the advantage of keeping *R* as a conceptual unity.

Storing return information in the activation record of the callee has a much more limited applicability: it supports only routine calling. The reason is simple: only the callee can get at the information and the only thing it can do with it is to use it to return to the caller. Hardware support is, however, much stronger for storing return information than for storing continuation information, and in virtually all implementations the call to a routine *S* stores return information in the activation record of *S*.

The administration part may contain another important entry: the "lexical pointer", also called "static link"; this is the frame pointer of the enclosing visible scope. The lexical pointer allows access to the local variables of the lexically enclosing routine and to its lexical pointer, as will be explained in Section 11.3.6. For some languages, no lexical pointer is needed. In C, for example, routines can only be defined on the top level, so a routine has only one enclosing visible scope, the global scope, in addition to its own local scope. The frame pointer of the global activation record is constant, though, and need not be stored; it can be incorporated into the machine addresses of the global items. The frame pointer of the local activation record is available as the current frame pointer FP.

Activation records can be allocated on a stack or on the heap. Stack allocation is cheaper but limits the usability of the activation records. As said before, we will try to use stack allocation as long as possible in our implementation.

11.3.3 Routines

A routine is just a piece of code, reachable through a pointer to its first instruction, its **code address**. When a routine *R* is called (invoked, activated) an activation record is created for it and the flow of control is transferred to its first instruction by setting the program counter to the routine code address; the routine is now **running**. When *R* calls a routine *S*, an activation record for *S* is created and the flow of control is transferred to the first instruction of *S*; *S* is now running and *R* is **suspended**. *R* is then the **parent** of *S*, and *S* is the **child** of *R*. When *S* finishes, it is **terminated**, and *R* is **resumed**. The **ancestors** of a routine *R* are defined as the parent of *R* and the ancestors of the parent of *R*. Routines that are running or suspended have activation records; they are called **active**. A routine can be active more than once simultaneously, in which case it is recursive; only one invocation can be running at any given time. When a routine returns, its activation record is removed. When the last activation record of a routine has been removed, the routine becomes **inactive** again.

The above is the behavior of the classical **subroutine**; there are several other kinds of routines, which exhibit additional features. The simplest is the **iterator**, a routine that can suspend itself temporarily and return to its parent without losing its activation record. This allows the iterator to continue where it suspended itself, when it is called again. As a result, the iterator can yield a succession of values; hence its name. The temporary return statement is usually called "suspend" or "yield".

An example application of an iterator in C notation is given in Figure 11.24. The iterator next_fibonacci() is started by the first call to it in the routine use_iterator(). After initializing a and b, the iterator immediately suspends itself on the yield a statement, yielding the value 0. The second call to next_fibonacci() continues just after the yield a statement, and again suspends immediately, yielding a 1. The third and further calls require the computation of further elements of the Fibonacci sequence, which are then yielded by the second yield b statement. (The construction for (;;) is C idiom for an infinite loop.) In languages that lack iterators, programmers usually implement them with global variables or by introducing additional parameters, as shown for example in Section 1.4.2, but the general implementation requires the retention of the activation record. A well-known example of an iterator is the UNIX routine getchar().

```
void use_iterator(void) {
    for (;;) {
        printf ("%d\n", next_fibonacci ());
    }
}

int next_fibonacci(void) {
    int a = 0; int b = 1;
    yield a;
    yield b;
    for (;;) {
        int tmp = a; a = b; b = tmp + b;
        yield b;
    }
}
```

Fig. 11.24: An example application of an iterator in C notation

A third variety of routines is the **coroutine**. Like the iterator it can suspend itself, but unlike the iterator suspending does not imply a return to the parent but rather transfers the control to a named coroutine, which is then resumed. This form of flow of control is often used in simulation programs, and was introduced by Simula 67 [41]. There the coroutine transfer statement is called “resume”. The statement resume(C) in a coroutine X leaves X temporarily and resumes the coroutine C at the same point the last resume statement in C left C temporarily. A simplified example application in a C-like notation concerns producer/consumer communication, as shown in Figure 11.25. The example shown there is overly simple, since in practice the resume(Consumer) statements may be hidden in routines called directly or indirectly by Producer(), and a subsequent resume(Producer) must continue inside those calls, with the complete environments intact. This requires retaining the activation records of those calls and all calls that lead to them.

Independently of these variants, routines can be global or nested. The code in a **global routine** only has access to global entities and to its own local entities. The code of a **nested routine**, declared on the same level as local variables, has access to these same global and strictly local environments, but also to entities declared in lexically intervening routines, as shown in Figure 11.26. Since j, k, and m reside in different activation records, providing access to them requires some thought.

```
char buffer[100];

void Producer(void) {
    while (produce_buffer()) {
        resume(Consumer);
    }
    empty_buffer();        /* signal end of stream */
    resume(Consumer);
}

void Consumer(void) {
    resume(Producer);
    while (!empty_buffer_received()) {
        consume_buffer();
        resume(Producer);
    }
}
```

Fig. 11.25: Simplified producer/consumer communication using coroutines

```
int i;

void level_0(void) {
    int j;

    void level_1(void) {
        int k;

        void level_2(void) {
            int m;

            ...             /* code has access to i, j, k, m */
            k = m;
            j = m;
        }

        ...                 /* code has access to i, j, k */
        j = k;
    }

    ...                     /* code has access to i, j */
}
```

Fig. 11.26: Nested routines in C notation

11.3.4 Operations on routines

In addition to declaring a routine, which specifies its name, parameter types, and return type and which is a compile-time action, several operations of increasing complexity can be performed on routines.

First of all, a routine can be **defined**. This differs from declaring a routine in that defining it supplies the code of the routine. Also, defining may be a run-time action: when the routine level_1() in Figure 11.26 is recursive, each recursive invocation defines a different routine level_2(), each having access to a different incarnation of variable k. A routine definition results in a defined routine, possibly represented at run time as a routine value.

No doubt the most important operation on a defined routine is **calling** it. Calling a defined routine creates an activation record for it and transfers control to its code. Details of routine calling, which include parameter passing, passing back a return value and returning from the routine, are covered in Section 11.4.2.2.

Once a defined routine is seen as a value, two more operations on it become possible: **passing it as a parameter** to a routine in a call, and **returning it as a value** from a routine call. An important difference between these two operations is that passing a routine value as a parameter introduces the value into a *smaller* pointer scope, whereas returning a routine as a value introduces the value into a *larger* scope. (Pointer scopes were treated in Section 11.2.3.2.) Consequently, returning routines as values is fundamentally more difficult to implement than passing them as parameters. It turns out that once we have implemented returning routines as values, we can also store them in arbitrary data structures, more in particular in global variables. Routines passed as parameters are important in the implementation of logic languages, routines stored in data structures are essential for functional languages. Both occur in some imperative languages, for example Icon and Algol 68.

Routine values resulting from these two operations must allow the same operations as any other defined routine. In particular, it must be possible to call them, to pass them on as parameters, and to return them as values.

A less frequent operation is **jumping out of a routine**. The destination of such a jump, also called a **non-local goto**, is a **non-local label**. A non-local label is a label in another routine. A variation on the code from Figure 11.26 is shown in Figure 11.27. The routine level_2() contains a non-local goto statement to label L_1 in routine level_1(). The goto L_1 statement terminates the activation of level_2() and transfers control to L_1 in routine level_1(); if level_2() is recursive, several invocations of level_2() will have to be terminated. If level_1() is recursive, each incarnation defines a different non-local label L_1.

In addition to being directly visible, the non-local label can also be passed to the running routine as a parameter. When the non-local label is returned as a value or stored in a data structure with a scope not smaller than that of the label itself, it is called a **continuation**. Continuations allow side branches of computations to be resumed, and support a remarkable programming paradigm [16].

Designing a non-local goto mechanism requires finding a representation for the non-local label that supports transfer of control, being passed as a parameter, and

```
void level_0(void) {

    void level_1(void) {

        void level_2(void) {

            ...
            goto L_1;
            ...
        }

        ...
    L_1:...
        ...
    }

    ...
}
```

Fig. 11.27: Example of a non-local goto

being returned as a value. Note that this representation does not need to be the same as that of a routine value.

The last operation on routines we will discuss in this section is **partial parameterization**. In partial parameterization, one or more actual parameters are supplied to a defined routine, but the routine is not called, even if all actual parameters have been supplied. Instead, a new defined routine results, with $n-m$ parameters, if the original routine had n parameters, of which m have been supplied; again, m can be equal to n. An example in C notation would be:

```
extern int add(int i, int j);     /* yields i + j */
int (*inc)(int i);                /* a routine variable inc */

int main(void) {
    ...
    inc = add(, 1);               /* supply the second parameter */
    ...
    printf ("%d\n", inc (5));
    ...
}
```

in which an external routine add(int, int) is parameterized with 1 as its second parameter, to yield a new one-parameter routine, inc(). No call is involved, just the creation of a new defined routine. The last line of the code calls inc() with one parameter, 5, to print the result 6.

There is a simpler form of partial parameterization, in which parameters are supplied one by one in order from left to right. This form is called “currying” and plays a important role in functional languages, as shown in Sections 12.1.7 and 12.4.4.

Since partially parameterized routines are just ordinary routines, it is important that such routines be implemented so that all operations available on routines are available on them. This includes calling, further parameterization, being a parameter, being a return value, and perhaps others.

11.3.5 Non-nested routines

We will first discuss the implementation of non-nested routines since they are simpler than nested routines. Non-nested routines can be implemented using stack allocation exclusively, except when partial parameterization is among the required operations.

A non-nested routine is represented at run time simply by the start address of its code. When called, a new activation record is stacked, as described in Section 11.4.2.2. If the code address is known at compile time, a routine call instruction can be used to transfer control to the routine; if the address results from run-time computation, an indirect routine call instruction must be used. A non-nested running routine has access to two environments only: the global data area and its own activation record. The global data area is addressed directly. The routine's own activation record is accessible through the frame pointer FP, which is kept pointing to the activation record of the running routine. Direct addressing and the frame pointer together provide access to the complete environment of a running non-nested routine.

A non-nested routine can be passed on as a parameter or returned as a value by just passing on or returning its code address. The same operations are possible on this passed or returned routine as on the original routine, since in both cases the code address is all that is needed.

Jumping out of a non-nested routine is not a natural concept, since besides the routine's own code there is no other syntactically visible code to jump to. Still, it is occasionally useful to terminate a running routine and transfer control to a marked code location in an ancestor routine. Two possible applications are: stopping a recursive search when an answer has been found; and handling exceptions (Section 11.4.3.2). The C programming language has the **setjmp/longjmp mechanism** for this. A call to the built-in routine setjmp(env) saves information about its code position and stack environment in a "jump buffer" pointed to by the parameter env, and returns 0; it marks a possible place in the execution of the program to which control may be transferred by performing the non-local goto. A later call to the built-in routine longjmp(env, val) restores the environment saved by the last call of setjmp(env), and returns from the call to setjmp() as if it returned val. This effectively implements a non-local goto, with the jump buffer representing the non-local label. A condition for the proper functioning of this mechanism is that the routine that called setjmp must still be active at the time the corresponding longjmp is called.

The mechanism is demonstrated in Figure 11.28. The routine find_div_7() implements a recursive search for a number divisible by 7, and is symbolic for any such search process. When a number divisible by 7 has been found, a longjmp() is performed to the label (*jmpbuf_ptr) passed as a parameter, otherwise the search continues with the next higher number. Without the longjmp call, the search recurses into infinity.

```
#include <setjmp.h>

void find_div_7(int n, jmp_buf *jmpbuf_ptr) {
    if (n % 7 == 0) longjmp(*jmpbuf_ptr, n);
    find_div_7(n + 1, jmpbuf_ptr);
}

int main(void) {
    jmp_buf jmpbuf;          /* type defined in setjmp.h */
    int return_value;

    if ((return_value = setjmp(jmpbuf)) == 0) {
        /* setting up the label for longjmp() lands here */
        find_div_7(1, &jmpbuf);
    }
    else {
        /* returning from a call of longjmp() lands here */
        printf ("Answer = %d\n", return_value);
    }
    return 0;
}
```

Fig. 11.28: Demonstration of the setjmp/longjmp mechanism

The driver establishes the "non-local label" by calling setjmp(); the actual label is not textually visible and is located after the else. The driver then initiates the search, starting at 1; the non-local label is passed as a parameter. When the solution is found, find_div_7 performs the non-local goto, which lands at the else branch.

Note that the traditional C programming technique of allocating the jmp_buf data structure among the global variables constitutes a violation of the pointer scope rules. The jump buffer will contain pointers to the activation record of the routine that fills it, so its scope is smaller than that of the global data area. If the jump buffer is filled in the routine main() in C the problem disappears, since there is no code on a global level that could access the jump buffer, but if the jump buffer is filled in a subroutine, a pointer scope violation can easily occur, resulting in a jump to a routine that has already been terminated. Passing the jump buffer address as a parameter to all interested routines as in Figure 11.28 solves the problem and is safe, but annoying.

Now that we have seen the feature and its use, we turn to its implementation. The implementation of setjmp(env) must at least save the frame pointer of its caller and its own return address in the jump buffer env. The implementation of longjmp(env, val) retrieves the destination activation record frame pointer and the return address from the jump buffer env. It then unstacks activation records until it finds the destination activation record and transfers to the return address. The implementation must also deliver val in the function result register.

Partial parameterization of non-nested routines cannot be implemented substantially more simply than that of nested ones. We will therefore postpone its discussion to the next section.

11.3.6 Nested routines

Not all operations on nested routines can be implemented using stack allocation for the activation record, but much can still be done to preserve the stack regime.

In addition to the usual code address, the routine descriptor used to represent a defined nested routine *R* must contain enough information to provide access to the data that are visible from the point of its definition. These are the constants, variables, parameters, routines, etc., of the lexically enclosing routines of *R* and reside in activation records of these routines. The straightforward way to provide this access is to include the frame pointer of the invocation of the routine in which *R* is defined (another solution, using "closures" is discussed below, in Section 11.3.6.5). This pointer is called the **lexical pointer** or **static link**. Referring to Figure 11.26, the descriptor of the routine level_2() consists of the code address of level_2() and a lexical pointer, the frame pointer of the enclosing routine level_1(). Figure 11.29 shows such a two-pointer routine descriptor.

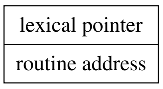


Fig. 11.29: A routine descriptor for a language that requires lexical pointers

Several points are worth noting here. The first is that it is conceptually convenient to imagine the definition of a routine to correspond to run-time code, which produces a correctly filled local routine descriptor in the activation record, just as the definition int i = 5; produces a correctly filled local integer variable. Figure 11.30 shows possible code for the construction of a routine descriptor for the routine level_2() from Figure 11.26. Further optimization may of course render the explicit construction of the routine descriptor superfluous, just as constant propagation can remove the allocation of i from the program if i turns out not to be modified anywhere. Also, the value level_2_as_a_value could be constructed on the fly when calling routine A() in Figure 11.30.

A second point is that if level_1() is recursive, each incarnation has a different activation record with a different address, so the routine descriptors for the different level_2()s in them differ, as they should.

```
void level_1(void) {
    int k;

    void level_2(void) {
        int l;

        ...
    }
    routine_descriptor level_2_as_a_value = {
        FP_of_this_activation_record(),   /* FP of level_1 () */
        level_2                           /* code address of level_2() */
    };

    A(level_2_as_a_value);                /* level_2() as a parameter */
}
```

Fig. 11.30: Possible code for the construction of a routine descriptor

Another point is that when nested routines have a two-pointer descriptor, it is next to necessary to use them for non-nested routines for reasons of uniformity. The top-level routine descriptors can get a null lexical pointer, since all data visible from outside a top-level routine is accessible by direct addressing and the lexical pointer will never be consulted.

Since the code of routine level_2() has access not only to the data of level_1() but also to those of level_0(), it would seem that supplying a lexical pointer to just the activation record of level_1() is not enough. We will see now, however, that it is.

11.3.6.1 Calling a nested routine

When a routine *R* defined by a two-pointer routine descriptor *D* is called, a new activation record is created; the present program counter PC, the frame pointer FP and the lexical pointer from *D* are stored in the administration area of the new activation record; FP is made to point to the new activation record; and control is transferred to the code address from *D*. See Figure 11.31.

As said, the point of having a lexical pointer is the access it allows to all lexically enclosing environments. We will now first see how the access to l and k in the statement k = l in Figure 11.26 can be obtained. The variable l is located in the activation record of the running routine, so it can be reached through the frame pointer: l is *(FP + offset(l)), where offset(*X*) is the offset of entry *X* from the frame pointer of the activation record. The variable k is located in the activation record of the immediately enclosing routine, which can be reached through the lexical pointer, which in turn can be found through the frame pointer: k is *(*(FP + offset(lexical_pointer)) + offset(k)) in routine level_2(). (Of course, k is *(FP + offset(k)) in routine level_1().) So the assignment k = l is translated to intermediate code as shown in Figure 11.32. The translation of the statement j = l is

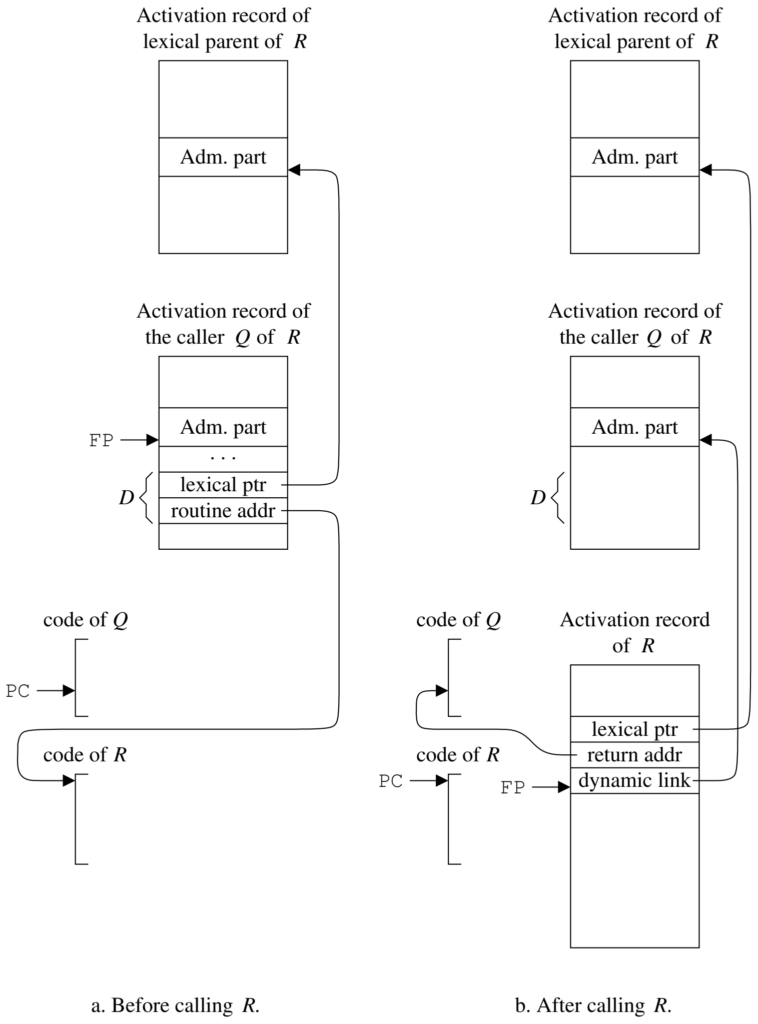


Fig. 11.31: Calling a routine defined by the two-pointer routine descriptor *D*

similar, except that j must be found by following the lexical pointer twice (Figure 11.32). These translations may look imposing, but BURS techniques can often find good code for such forms, by exploiting advanced addressing modes.

We see that storing the lexical pointer to the activation record of the lexically enclosing routine in the activation record of the running routine builds a linked list of those activation records that hold the data visible from the running routine. The length of the list is equal to the lexical nesting depth of the running routine.

The number of times the lexical pointer must be followed to reach an entry in a routine *R* from a routine *S* is equal to the difference in lexical nesting depth be-

```
*(
    *(
        FP
        +
        offset ( lexical_pointer )
    )
    +
    offset (k)
) =
*(FP + offset( l ))
```

Fig. 11.32: Intermediate code for the non-local assignment k = l

```
*(
    *(
        *(  FP
            +
            offset ( lexical_pointer )
        )
        +
        offset ( lexical_pointer )
    )
    +
    offset ( j )
) =
*(FP + offset( l ))
```

Fig. 11.33: Intermediate code for the non-local assignment j = l

tween *S* and *R*. It is therefore convenient to represent local addresses in the compiler as pairs of nesting difference and offset; since the nesting difference cannot be negative, a value of −1 can be used to code direct addressing. So, inside routine level_2(), l is represented as [0, offset(l)], k as [1, offset(k)], j as [2, offset(j)], and i as [−1, offset(i)]. Note that the nesting difference is a compile-time constant.

11.3.6.2 Passing a nested routine as a parameter

Passing a nested routine as a parameter is simple now: just pass the two-pointer descriptor. No matter to what static or dynamic depth the routine level_2() is passed on, when it is finally called the above calling scheme constructs an activation record with a lexical pointer that indicates the activation record of level_1() and thus restores the proper environment for a call of level_2(); see Figure 11.34.

Since the two-pointer routine descriptor contains a pointer of possibly limited scope, the lexical pointer, we have to consider the pointer scope rules. The scope of the routine descriptor is the same as that of the activation record in which the routine was declared. When passing the routine descriptor to a child routine, it is passed into an environment of smaller scope, so no scope violation can occur, regardless of

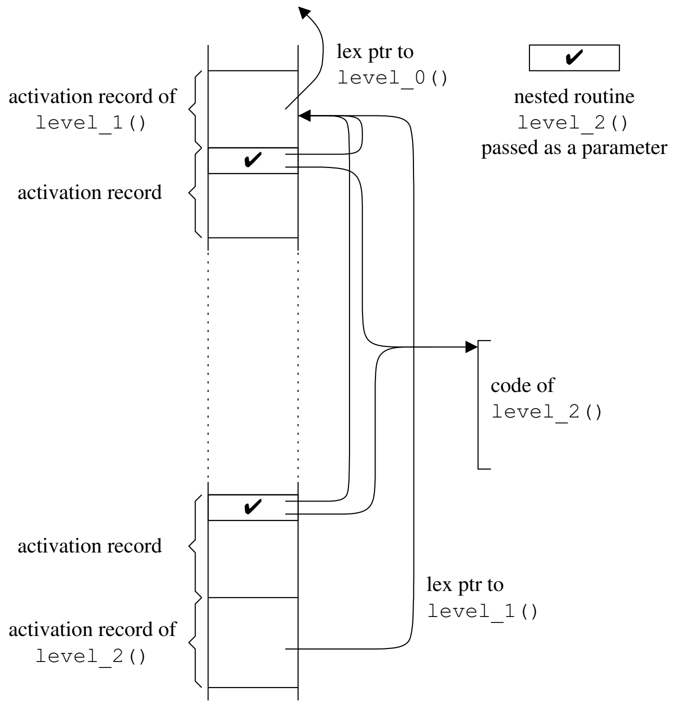


Fig. 11.34: Passing a nested routine as a parameter and calling it

whether we use a heap or a stack for the activation records.

11.3.6.3 Returning a nested routine as a value

Returning a nested routine as a value is equally simple: just return the two-pointer descriptor. Now, however, the two-pointer descriptor is passed to an environment of potentially larger scope, so there is the danger of a pointer scope violation. The violation will indeed occur under a stack regime: when routine level_1() returns routine level_2() as a value to its parent level_0(), and this parent calls the returned routine, the call will construct an activation record for level_2() whose lexical pointer refers to the activation record of level_1(), which is long gone! We see that returning a nested routine as a value is incompatible with the stack regime; it requires the activation records to be allocated on the heap.

Heap allocation indeed solves the problem. Since all environments have the same infinite scope, no data is introduced into environments of larger scope, and no pointer scope violation can occur. More in detail, the activation record of the call of routine level_1() in our example will be retained automatically after the call has terminated, since it is still accessible from the program data area: FP, which is in the root set, points to the activation record of level_0(), which contains the routine value level_2(), which contains the frame pointer of level_1() as the lexical pointer. Such operations may seem weird in a C-like language, but are commonplace in functional languages, and in some imperative languages, for example Icon and some variants of Algol 68.

11.3.6.4 Jumping out of a nested routine

The main operation a non-local label in a given routine invocation *I* must support is the non-local goto. Performing a non-local goto to this label terminates zero or more routine invocations until routine invocation *I* surfaces and then transfers control to the local label.

This can be implemented by representing the non-local label as a two-pointer descriptor containing the frame pointer of the routine invocation that holds the label, and the code address of the label. The implementation of a non-local goto to a non-local label *L* must then travel back through the activation record chain as defined by the dynamic links, until it finds the activation record with a frame pointer that is equal to that in *L*. Each activation record met on the way must be released, since the activation of the corresponding routine is terminated implicitly by the non-local goto. Once the proper activation record has been found, FP is made to point to it and execution proceeds at the code address indicated in *L*.

If routine level_1() in Figure 11.27 is recursive, each invocation defines a different label L_1. Their representations differ by pointing to different activation records; performing non-local gotos on them will terminate different numbers of active routines.

The two-pointer non-local label descriptor can be passed as a parameter or returned as a value in the same way as a routine descriptor can, and the same pointer scope considerations hold. The two are, however, fundamentally different: a routine descriptor holds a pointer to an activation record that will be the parent of a new running routine and a code address to be jumped to by a routine call instruction, a non-local label descriptor holds a pointer to an activation record that will itself be running again and a code address to be jumped to by a jump instruction.

As with routine definitions, it is often convenient to allocate the non-local label descriptor as a constant entry in the activation record, as shown in Figure 11.35. Again, however, values like L_1_as_a_value may be constructed on the fly.

11.3.6.5 Partial parameterization

The data structure representing a partially parameterized routine must be capable of holding an indeterminate number of actual parameters, so the above two-pointer descriptor does not suffice and will have to be extended. The extension consists of space for all its parameters, plus a mask indicating which parameters have al-

```
void level_1(void) {
    non_local_label_descriptor L_1_as_a_value = {
        FP_of_this_activation_record(),   /* FP of level_1 () */
        L_1        /* code address of L_1 */
    };

    void level_2(void) {

         ...
        non_local_goto(L_1_as_a_value); /* goto L_1; */
         ...
    }

     ...
L_1 :...
     ...
}
```

Fig. 11.35: Possible code for the construction of a label descriptor

ready been filled in. Figure 11.36 shows the result of partially parameterizing a 5-parameter routine with its second and fourth parameter values.

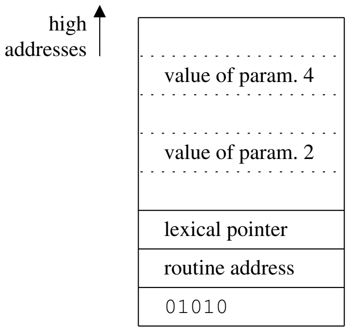


Fig. 11.36: A closure for a partially parameterized routine

This representation of a partially parameterized routine is called a **closure**; unfortunately it is the same term as used in "transitive closure" and "closure algorithm", with which it has nothing to do. The closure can be allocated on the stack or on the heap, with different consequences for the pointer scope rules.

Once we have this representation, implementing the desired operations on it is simple.

Further parameterization of a closure is implemented by scanning the mask to find the locations of the free parameters and filling them in. The required type checking has already been done since the corresponding routine just has a new routine type with a subset of the original parameters. If the language specifies that parameters are always supplied one at the time from left to right, the mask can be replaced by a counter and finding the location of the first free parameter is trivial. This "currying" style of partial parameterization occurs in many functional languages, and is explained in more detail in Section 12.1.7.

Once all parameters of the routine have been supplied, it can, but need not be invoked, since a fully parameterized routine can be treated as a parameterless routine (see page 563).

The invocation of a fully parameterized routine can be done remarkably efficiently: allocate room for the activation record and block-copy the closure into it. If the layout is chosen properly, the parameters and the lexical pointer are already in place, and the mask can be overwritten by the dynamic link; such a layout is already shown in Figure 11.36.

Passing a routine represented by a closure as a parameter or returning it as a value can be implemented in the obvious way: pass or return a copy of the closure. Since the closure contains the required lexical pointer, it can be invoked in the same way as described above. This implementation requires that *all* routines be treated as partially parameterized and closures must be used everywhere instead of two-pointer routine descriptors.

11.3.6.6 Discussion

In this subsection we have considered representations for nested routines and implementations for the operations on them. Some of these operations, notably passing them as parameters and returning them as values, give rise to new routines and we have been careful to ensure that our implementations of the operations also work on them.

Most operations on routines and non-local labels are compatible with a stack regime, except returning them as values, which requires heap-allocated activation records. Since heap allocation of the activation records slows down the calling mechanism, it is important to seek ways to avoid it and maintain a stack regime. We will now discuss such a way.

11.3.7 Lambda lifting

The problem with two-pointer routine descriptors is that they contain a lexical pointer, which points into the stack and which reduces its pointer scope. As a result, two-pointer routine descriptors cannot be moved around freely, which in turn limits their usability. One simple solution was given above: allocate all activation records on the heap. We will now consider another simple solution: do away with lexical pointers and make all routines global. This immediately reintroduces the original problem: how to gain access to the non-local data; but we now give a new answer: pass pointers to them as parameters.

Figure 11.37 shows the result of this transformation on the routines of Figure 11.26. At the top we see the innermost routine level_2(), which used to have access to two non-local non-global variables, j and k, and which now has two pointer parameters. When called (in level_1()), the actual parameters are supplied from j, which level_1() has received as a parameter, and the address of k. A similar explanation applies to the call of level_1() in level_0(). It is clear that the two assignments *k = l and *j = l assign the value of l to the proper locations.

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...    /* code has access to i, *j, *k, l */
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int k;

    ...    /* code has access to i, *j, k */
    level_2(j, &k);    /* was: level_2(); */
    A(level_2);    /* level_2() as a parameter: */
        /* this is a problem */
}

void level_0(void) {
    int j;

    ...    /* code has access to i, j */
    level_1(&j);    /* was: level_1(); */
}
```

Fig. 11.37: The nested routines from Figure 11.26 lambda-lifted (in C notation)

The transformation shown here is called **lambda lifting**. The name derives from the lambda expressions in Lisp and other functional languages that are lifted to global level by it, but the technique has been in use with C programmers for ages.

Lambda lifting has effectively rid us of two-pointer routine descriptors with their limited scopes, but the moment we try to pass a lifted routine as a parameter, we run into a new problem: how do we pass on the extra parameters that result from the lambda lifting? When passing level_2() as a parameter, how do we pass the j and &k that go with it? There is a stunning answer to this question: use partial parameterization! Rather than passing the code address of level_2(), we pass a closure *C* containing the code address, j, and &k. Note that these closures, unlike the earlier closures, do not contain lexical pointers. When the routine parameter corresponding to *C* is finally called, the run-time system recognizes it as a closure and performs a closure invocation.

The scope of closure *C* is determined by the scope of &j, which, unfortunately is again that of the activation record of the lexically enclosing routine, level_1(). This still prevents the closure from being returned as a value, due to pointer scope problems. This problem is solved by another drastic measure: all local data that is used non-locally is allocated on the heap and the corresponding local entries are replaced by pointers. Figure 11.38 shows the final result. In particular, the closure passed in the call of A() has infinite scope: it contains a code address and two heap pointers. Thus, the closure can be moved to any environment and called wherever needed. The activation record of level_1() may long have disappeared; the required "locals" in it still exist on the heap.

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...                           /* code has access to i, *j, *k, l */
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int *k = (int *)malloc(sizeof (int));

    ...                           /* code has access to i, *j, *k */
    level_2(j, k);                /* was: level_2(); */
    A(closure(level_2, j, k));    /* was: A(level_2); */
}

void level_0(void) {
    int *j = (int *)malloc(sizeof (int));

    ...                           /* code has access to i, *j */
    level_1(j);                   /* was: level_1(); */
}
```

Fig. 11.38: Lambda-lifted routines with additional heap allocation in C notation

We have now achieved an implementation in which the stack mechanism is used for the routine invocation administration and the strictly local variables, and in which variables that are used non-locally are allocated on the heap. Such an implementation is advantageous for languages in which efficient routine calling and free movement of routine variables are very important. The implementation is especially profitable for languages that require partial parameterization anyway; in short, for implementing functional languages and advanced imperative languages. The main property of the implementation is that routines are represented by closures that do not carry lexical pointers; this gives them the opportunity to acquire infinite scope.

11.3.8 Iterators and coroutines

The above techniques give us enough material to implement both iterators and coroutines in a simple way. The two implementations are almost identical. The invocation of an iterator or coroutine creates an activation record in the usual way, and its address is stored in a variable, allocated for the purpose; the activation record will have to be allocated on the heap, unless the functionality of the iterator or coroutine is restricted severely. When either is suspended, continuation information is stored in that activation record. The iterator returns temporarily to its caller, whose activation record can be found by following the dynamic link; the coroutine transfers to another coroutine, whose activation record can be found in its coroutine variable. When the iterator or coroutine is resumed the continuation information is retrieved and acted upon. When the iterator or coroutine terminates, its activation record is deleted, and the iterator or coroutine variable zeroed.

This concludes our discussion of non-nested and nested routines and the implementation of several operations applicable to them. The basic data structures in all the implementations are the activation record and the routine representation. The latter exists in two variants, the two-pointer routine descriptor and the closure. The detailed allocation design decisions are based on the pointer scope rules explained in Section 11.2.3.2.

11.4 Code generation for control flow statements

In Chapters 7 and 9 we discussed code generation for expressions and basic blocks. In this section, we will concentrate on code generation for statements that affect the flow of control, and thus demarcate the basic blocks. Three levels of flow of control can be distinguished:

- local flow of control, which determines the statement inside a routine or method to be executed next (Section 11.4.1);
- routine calls and method invocations, which perform the parameter transfer and flow-of-control manipulation needed to activate a new routine (Section 11.4.2);
- non-local jumps, which transfer the flow of control out of the currently running routine into an ancestor routine (Section 11.4.2.3).

We assume that all source code expressions, with one exception, have already been evaluated and that the results have been stored in an appropriate place, usually a register. The exception is formed by Boolean expressions used for flow control, for example the control expressions in if-statements; they are treated separately in Section 11.4.1.1. In addition, we assume the existence of a mechanism in the compiler for allocating temporary variables and labels. This description is intentionally not very precise; details always depend on the source and target languages.

To describe the code generation for the flow of control, we will use the four statement types below, written in a Pascal-like notation. Each has a straightforward equivalent on virtually all processors.

- A simple goto statement: GOTO *label*; the address of the destination is the constant value *label*.
- An indirect goto statement: GOTO *label_register*; the address of the destination is the contents of *label_register*.
- A conditional goto statement, in two forms:
  IF *condition_register* THEN GOTO *label*
  and
  IF NOT *condition_register* THEN GOTO *label*.
- An assignment statement: *destination* := *source*, used to compute temporary values needed for the flow of control.

We will sometimes write simple expressions where registers appear in the above statements, especially when these expressions derive from the code generation mechanism itself rather than from the source code; this increases the readability of the code samples.

Roadmap

11.4.1 Local flow of control

The two main mechanisms for influencing the local flow of control in imperative and object-oriented languages are “selection” and “repetition”. Selection causes a piece of code to be selected for execution, based on the value of some expression. Repetition causes a piece of code to be executed zero or more times, based on the value of some expression or expressions. More often than not these expressions are Boolean expressions, and in many cases it is useful to translate them in special ways. We will therefore first consider code generation for Boolean expressions used in controlling program execution, and then turn to code for selection and repetition statements.

11.4.1.1 Boolean expressions in flow of control

Fundamentally, Boolean expressions are no different than other expressions: evaluating one yields a Boolean value. Most Boolean expressions are, however, used to affect the flow of control rather than to produce a value; we will call Boolean expressions used in that way **Boolean control expressions**. There are two reasons to treat Boolean control expressions specially.

The first reason has to do with two properties of machine instructions. Boolean expressions often consist of comparisons and the comparison instructions of most machines produce their results in special condition registers in a special format rather than as 0/1 integer values on the stack or in a register; so an additional conversion is required to obtain a genuine Boolean value. And the most usual way of affecting the flow of control is by using a conditional jump, and the machine instructions for conditional jumps base their decisions on condition registers rather than on 0/1 integer values; so jumping on a Boolean value requires an additional conversion of the Boolean value to a value in a condition register. Obviously, the naive code sequence for Boolean control expressions

```
/* code for the Boolean expression: */
comparison code, yielding a condition value
conversion from condition value to Boolean
/* code for the conditional jump: */
conversion from Boolean to condition value
jump on condition value
```

is to be avoided.

The second reason for treating Booleans specially is related to a property of some programming languages. Several programming languages (for example C, Ada, Java) feature lazy Boolean operators, operators that evaluate operands only when their value is needed. Examples are the && and || operators in C. Such operators do not fit the translation model discussed in Sections 7.5.2.1 and 7.5.2.2: expr_1 && expr_2 cannot be translated as

```
code to compute expr1 in loc1
code to compute expr2 in loc2
code for the && operator on loc1 and loc2
```

since that would result in the unconditional evaluation of both expressions. Instead, code intermingled with conditional jumps must be generated; again these conditional jumps react to values in condition registers. In short, Boolean control expressions are tightly interrelated with conditional jumping.

This relationship can be exploited conveniently when we know the labels to which control must be transferred when the Boolean expression yields true or false, before we generate the code. We can then use a code generation technique like the one shown in Figure 11.39. The procedure *GenerateCodeForBooleancontrolexpression* gets two parameters, *TrueLabel* and *FalseLabel*, in addition to the usual *Node* pointer. A special value *NoLabel* is available for these parameters, to indicate that control must continue at the end of the expression: the control must "fall through" to the end. We assume a single condition register here, although most machines have several of them, with assorted semantics.

```
procedure GenerateCodeForBooleanControlExpression (Node, TrueLabel, FalseLabel):
    select Node.type:
        case ComparisonType:            −− <, >, ==, etc. in C
            GenerateCodeForComparisonExpression (Node.expr);
            −− The comparison result is now in the condition register
            if TrueLabel ≠ NoLabel:
                Emit ("IF condition_register THEN GOTO" TrueLabel);
                if FalseLabel ≠ NoLabel:
                    Emit ("GOTO" FalseLabel);
            else −− TrueLabel = NoLabel:
                if FalseLabel ≠ NoLabel:
                    Emit ("IF NOT condition_register THEN GOTO" FalseLabel);
        case LazyAndType:               −− the && in C
            −− Create EndLabel to allow left operand fall-through:
            EndLabel ← NewLabel ();
            if FalseLabel = NoLabel:
                −− The lazy AND should fall through on failure
                LeftOperandFalseLabel ← EndLabel;
            else −− The lazy AND should fail to the original FalseLabel:
                LeftOperandFalseLabel ← FalseLabel;
            GenerateCodeForBooleanControlExpression
                (Node.left, NoLabel, LeftOperandFalseLabel);
            GenerateCodeForBooleanControlExpression
                (Node.right, TrueLabel, FalseLabel);
            Emit ("LABEL" EndLabel ":");
        case LazyOrType:                −− the || in C
            . . .
        case NegationType:              −− the ! in C
            GenerateCodeForBooleanControlExpression
                (Node.left, FalseLabel, TrueLabel);
```

Fig. 11.39: Code generation for Boolean expressions

If the node represents a (numeric) comparison operator, we generate code for this operator; this leaves the result in the condition register. Then, depending on the presence or absence of the true and false labels, we generated zero, one or two jump instructions.

The use of the value *NoLabel* is shown in the entry for the lazy && operator. First we generate code for the left operand of the && operator, such that when it succeeds, control falls through to reach the code of the right operand of the && operator. What happens when the left operand fails is more complicated. We cannot just transfer control to the *FalseLabel* since it could be *NoLabel*, in which case we have to lead the control on to the end of the code generated for the &&. The auxiliary label *LeftOperandFalseLabel* take care of this.

Similar entries can be constructed for the || and ?: operators of C. The last entry in Figure 11.39 shows that the implementation of the negation operator comes free of charge: we just swap the true and false labels.

As an example, the call

```
GenerateCodeForBooleanControlExpression (Parse ("i > 0 && j > 0"), NoLabel, Else label)
```

in which we assume that *Parse(string)* produces the parse tree for *string*, yields the code sequence

```
Compare_greater i, 0
IF NOT condition_register THEN GOTO Else label
Compare_greater j, 0
IF NOT condition_register THEN GOTO Else label
```

There are also occasions when we have to construct a genuine Boolean value, for example to assign it to a variable or to pass it as a parameter. We can then use conversion instructions if the target machine has them, or use the above scheme to produce code to jump to places where the proper values are constructed.

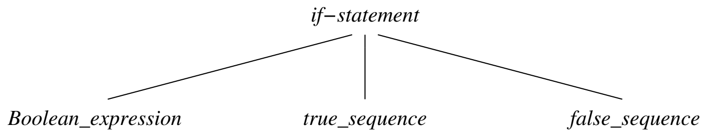
11.4.1.2 Selection statements

The two most common selection statements are the if-statement and the case statement. The if-statement selects one of two statement sequences (one of which may be absent), based on the value of a Boolean expression; the case statement (also sometimes called a “switch statement” or a “selection statement”) selects one out of several statement sequences, based on the value of an integer or enumeration expression.

The if-statement The general form of an **if-statement** is:

```
IF Boolean_expression THEN true_sequence ELSE false_sequence END IF;
```

which results in the AST



Code generation for an if-statement is simple: create two new labels, false_label and end_label, and generate the following code:

```
    BooleanControlCode (Boolean_expression, 0, false_label)
    code for true_sequence
    GOTO end_label;
false_label:
    code for false_sequence
end_label:
```

where *BooleanControlCode* is the code generated by the procedure *GenerateCodeForBooleanControlExpression* with the parameters shown (*NoLabel* is represented by 0).

If the else-part is absent (or empty), the false_label is not needed and we generate

```
    BooleanControlCode (Boolean_expression, 0, end_label)
    code for true_sequence
end_label:
```

The case statement We will consider **case statements** of the form:

```
CASE case_expression IN
    $I_1$: statement_sequence$_1$
    . . .
    $I_n$: statement_sequence$_n$
    ELSE else-statement_sequence
END CASE;
```

where I_1 ... I_n are **case labels** —integer or enumeration values representing compile—time constants. The expression *case_expression* must be evaluated. If its value is equal to one of the values $I_1, \ldots, I_n$, the corresponding statement sequence is executed. If not, the statement sequence of the else-part is executed.

There are many code generation schemes for case statements and it is the task of the compiler to select an efficient one for the statement at hand. The choice depends on the number of case entries, the range (or reach) of the case labels, and on the density of the case labels within the range.

The following is a simple scheme that works well when there are only a few case entries (n being small, say 10 or less). First, $n+2$ labels are allocated: label_1 through label_n, label_else, and label_next. Also, a temporary variable is allocated for the case expression. Then, the code of Figure 11.40 is generated. This scheme implements a linear search through all the case labels. Note that we allocate a temporary variable for the case expression. Usually, the language manual specifies that the case expression must be evaluated only once, and, even if the language manual does not, it may still be more efficient to do so.

```
    tmp_case_value := case_expression;
    IF tmp_case_value = $I_1$ THEN GOTO label_1;
    . . .
    IF tmp_case_value = $I_n$ THEN GOTO label_$n$;
    GOTO label_else;                  −− or insert the code at label_else
label_1:
    code for statement_sequence$_1$
    GOTO label_next;
    . . .
label_$n$:
    code for statement_sequence$_n$
    GOTO label_next;
label_else:
    code for else-statement_sequence
label_next:
```

Fig. 11.40: A simple translation scheme for case statements

The execution time of the above scheme is linear in n, the number of cases in the case statement. Case selection in constant time is possible using a **jump table**, as follows. First the compiler computes the lowest case label I_{low} and the highest case label I_{high}. Then the compiler generates a table of $I_{high} - I_{low} + 1$ entries, to be indexed with indices ranging from 0 to $I_{high} - I_{low}$. The entries in this table are code labels: label_k for an entry with index $I_k - I_{low}$, for k ranging from 1 to n, and label_else for all others. Finally the following code is generated:

```
tmp_case_value := case_expression;
IF tmp_case_value < $I_{low}$ THEN GOTO label_else;
IF tmp_case_value > $I_{high}$ THEN GOTO label_else;
GOTO table [tmp_case_value – $I_{low}$];
```

If $I_{high} - I_{low}$ is much larger than n, many of the jump table entries contain label_else, and the table may be deemed too space-inefficient. In that case, the case labels can be organized into a balanced binary tree, in which each node of the tree represents one case label I, the right branch indicates a subtree with case labels larger than I, and the left branch indicates a subtree with case labels smaller than I. For each node node_k in the binary tree, the following code is generated:

```
label_k:
    IF tmp_case_value < $I_k$ THEN
        GOTO label of left branch of node_k;
    IF tmp_case_value > $I_k$ THEN
        GOTO label of right branch of node_k;
    code for statement_sequence$_k$
    GOTO label_next;
```

If the left branch and/or the right branch does not exist, the corresponding GOTO is replaced by GOTO label_else.

Many more advanced translation schemes for case statements exist. Several translation schemes for the case statement were analyzed and compared by Sale [245]. Very sophisticated techniques for producing good code for the case statement are described by Hennessy and Mendelsohn [120], Bernstein [36], and Kannan and Proebsting [143].

11.4.1.3 Repetition statements

The most common repetition statements are the while statement and the for-statement. The while statement executes a statement sequence an indeterminate number of times (including 0 times), as long as the while expression remains fulfilled. The for-statement executes a statement sequence a fixed number of times.

The while statement The **while statement**

```
WHILE Boolean_expression DO statement_sequence END WHILE;
```

can be processed by allocating two labels: end_label and test_label, and generating the following code:

```
test_label:
    BooleanControlCode (Boolean_expression, 0, end_label);
    code for statement_sequence
    GOTO test_label;
end_label:
```

In many cases, however, the following scheme results in more efficient code: allocate two labels: sequence_label and test_label, and generate the following code:

```
    GOTO test_label;
sequence_label:
    code for statement_sequence
test_label:
    BooleanControlCode (Boolean_expression, sequence_label, 0);
```

This scheme is usually more efficient when there are several iterations, because it only executes a single conditional jump instruction per iteration, whereas the first scheme executes a conditional jump instruction and an unconditional jump instruction per iteration. If *Boolean_expression* evaluates to false the first time, there will be no iterations, and the first scheme is more efficient.

Which scheme is more efficient also depends on the target processor. Note that the branch in the conditional goto statement of the first scheme is actually taken only once. On many processors, in particular processors that maintain an instruction look-ahead cache, a conditional branch instruction is more expensive when the branch is taken than when it is not, because when it is, the instruction look-ahead cache must be flushed. On the other hand, some modern processors perform look-ahead at the target of the branch instruction as well, and/or have a sophisticated branch prediction mechanism.

The for-statement We will first consider the following type of **for-statement**:

```
FOR i IN lower_bound..upper_bound DO
    statement_sequence
END FOR;
```

where i is the controlled variable of the for-statement; the implicit step size is 1; and *lower_bound* and *upper_bound*, both inclusive, are to be evaluated once upon starting the for-statement. Code generation for a for-statement is quite tricky, because care must be taken that the controlled variable of the for-statement does not cause overflow.

The intuitive approach is to allocate a temporary variable tmp_ub for the upper bound, and generate the following code:

```
i := lower_bound;
tmp_ub := upper_bound;
WHILE i <= tmp_ub DO
    code for statement_sequence
    i := i+1;                      -- WRONG: may cause overflow
END WHILE;
```

where the while statement is handled as described above. Unfortunately, this scheme will not always work. In particular, the computation of *upper_bound* may produce the largest value representable for the type of the controlled variable i. After a while i will reach the value of tmp_ub. Then, on machines that detect overflow, the increment of the controlled variable will cause an exception. On machines that do not detect overflow, the for-statement will never terminate, because i can never become larger than the largest value representable for its type, and can thus never become larger than tmp_ub.

Therefore, the loop termination test must compare the controlled variable i with tmp_ub for equality, and be executed after the statement sequence, but before the increment of the controlled variable. However, moving the loop termination test to the end of the statement sequence means that another test is required to determine if the loop should be entered at all. This leads to the following, improved, scheme (with the temporary variable tmp_ub allocated as above, and labels loop_label and end_label):

```
    i := lower_bound;
    tmp_ub := upper_bound;
    IF i > tmp_ub THEN GOTO end_label;
loop_label:
    code for statement_sequence
    IF i = tmp_ub THEN GOTO end_label;
    i := i + 1;
    GOTO loop_label;
end_label:
```

In this generation scheme, the first IF clause makes sure that the statement sequence will never be executed when *lower_bound* is larger than *upper_bound*. Also, the controlled variable i will never be incremented beyond *upper_bound*, thus preventing overflow and its consequences.

An issue that we have ignored until now is what value the controlled variable i should have after the loop is finished. In some languages, the for-loop declares the controlled variable implicitly, so it no longer exists after the loop. Some other languages do not specify what value the controlled variable should have afterwards, or explicitly specify that it is not specified (so that programs that depend on it are erroneous). When the language manual does specify the value, the implementation should of course follow the manual, if necessary by adding the proper assignment at the end_label.

Many languages offer more general forms of for-statements, for example providing an explicit step size. An explicit step size causes additional complications, as is illustrated by the following for-statement:

```
FOR i IN 1..6 STEP 2 DO ... END FOR;
```

The complication here is that the controlled variable never becomes exactly equal to the upper bound of the for-statement. Therefore, the scheme described above fails miserably. With an explicit step size, we cannot compare the upper bound and the controlled variable for equality, and, as we have seen above, we cannot compare for greater/smaller either, because of possible overflow problems.

A solution lies in computing first the number of times the loop will be executed. An extra temporary variable is needed to hold this number. Note that this temporary loop count must have a range that is large enough to represent the difference between the maximum value and the minimum value representable in the type of the controlled variable. For example, if the controlled variable is of type integer, the value MAX(integer) − MIN(integer) must be representable in this temporary variable. Remarkably, this can be accomplished by making the loop count an unsigned integer of the same size as the integer: if the representable values of an integer range from -2^n to 2^n-1, then the representable values of an unsigned integer of the same size range from 0 to $2^{n+1}-1$, which is exactly large enough. This does not work for a step size of 0. Depending on the language this exception can be detected at compile-time. Also, if the step size expression is not a constant, the step size needs a temporary variable as well.

All of this leads us to the scheme of Figure 11.41. Note that, although we no longer use the controlled variable to detect whether the for-statement is finished, we still have to keep its value up-to-date because it may be used in the *statement_sequence* code. Also note that the compiler can switch to the simpler and more efficient scheme discussed earlier when it can determine that the step size is 1.

```
    i := lower_bound;
    tmp_ub := upper_bound;
    tmp_step_size := step_size;
    IF tmp_step_size = 0 THEN
        ... probably illegal; cause run-time error ...
    IF tmp_step_size < 0 THEN GOTO neg_step;
    IF i > tmp_ub THEN GOTO end_label;
    -- the next statement uses tmp_ub − i
    --    to evaluate tmp_loop_count to its correct, unsigned value
    tmp_loop_count := (tmp_ub − i) DIV tmp_step_size + 1;
    GOTO loop_label;
neg_step:
    IF i < tmp_ub THEN GOTO end_label;
    -- the next statement uses i − tmp_ub
    --    to evaluate tmp_loop_count to its correct, unsigned value
    tmp_loop_count := (i − tmp_ub) DIV (−tmp_step_size) + 1;
loop_label:
    code for statement_sequence
    tmp_loop_count := tmp_loop_count − 1;
    IF tmp_loop_count = 0 THEN GOTO end_label;
    i := i + tmp_step_size;
    GOTO loop_label;
end_label:
```

Fig. 11.41: Code generated for a general for-statement

Sometimes, what looks like a for-statement actually is not. Consider, for example, the C for-loop of Figure 11.42. Here, expr1 and expr3 may contain any expression, including none at all. The expression expr2 may be absent (in which case we have an infinite loop), but if it is present, it must return a value of a type that is allowed in a condition context. If expr2 is present, the C for-loop is almost, but not quite, equivalent to the while loop of Figure 11.43; for a difference see Exercise 11.29.

```
for (expr1; expr2; expr3) {
    body;
}
```

Fig. 11.42: A for-loop in C

```
expr1;
while (expr2) {
  body;
  expr3;
}
```

Fig. 11.43: A while loop that is almost equivalent to the for-loop of Figure 11.42

Code generation for repetition statements is treated in depth by Baskett [32].

Optimizations for repetition statements As can be seen from the above, the administration overhead for for-loops can be considerable. An effective optimization that reduces this overhead, at the cost of increasing code size, is loop unrolling. In **loop unrolling**, the body of the loop is replaced by several copies of it, and the administration code is adjusted accordingly. For example, the loop

```
FOR i := 1 TO n DO
     sum := sum + a[i];
END FOR;
```

can be replaced by the two for-loops shown in Figure 11.44. In this example, we have chosen an **unrolling factor** of 4. Note that, in general, we still need a copy of the original loop, with adjusted bounds, to deal with the last couple of iterations. If the bounds are compile-time constants, it may be possible to avoid this copy by choosing the unrolling factor to be a divisor of the loop count.

This optimization is particularly effective when the body is small, so the unrolled loop still fits in the instruction cache. It may also be used to increase the size of the basic block, which may improve chances for other optimizations.

It is sometimes useful to generate separate code for the first or last few iterations of a loop, for example to avoid a null pointer check in every iteration, to ensure memory accesses in the main loop are aligned, or to avoid special code in the main loop for the final, potentially partial, iteration of the loop. This optimization is called **loop peeling**.

```
FOR i := 1 TO n–3 STEP 4 DO
    –– The first loop takes care of the indices 1 .. (n div 4) * 4
    sum := sum + a[i];
    sum := sum + a[i+1];
    sum := sum + a[i+2];
    sum := sum + a[i+3];
END FOR;

FOR i := (n div 4) * 4 + 1 TO n DO
    –– This loop takes care of the remaining indices
    sum := sum + a[i];
END FOR;
```

Fig. 11.44: Two for-loops resulting from unrolling a for-loop

11.4.2 Routine invocation

Until now we have discussed code generation for statements that affect the local flow of control. In this section, we will discuss routine calls and object method invocations. Execution of a *routine call* transfers the flow of control to the start of the called routine. The called routine will eventually return the flow of control to just after the routine call. Two important issues in calling a routine are routine identification —finding out which routine to call— and how to perform the call and return. A third operation connected with the flow of control in and around routines is the non-local goto statement, which terminates the running routine and transfers control to a labeled code location in an ancestor of the running routine. This deviates from the simple call–return scheme described above, since the call is not terminated by the expected return.

In object-oriented languages, we invoke methods on objects. The effect of a method invocation with regard to the flow of control is identical to that of a routine call. Method invocation and routine call differ in degree in that the routine to be called is almost always determined statically and the method to be invoked is often only determined at run time, by using a dispatch table. An additional difference is that a method has direct access to the fields of the object. This access is implemented by passing a pointer to the object as an additional parameter, as discussed in Section 11.2.9. We will now first consider what to call and then how to call it.

11.4.2.1 Routine identification—what to call

Before we can translate a routine call, the routine must be identified. Usually this has already been done during semantic checking. A routine name may be overloaded, but if the input program is correct, the language rules allow the compiler to identify a single routine to be called.

In languages that allow routine variables, the routine to be called may be the result of an expression. In this case, the compiler will not be able to identify the routine to be called, and must produce code to evaluate the expression. This should result in a (run-time) routine value, as discussed in Section 11.2.8; the routine value can then be called through an (indirect) routine call, for which an instruction is available on all reasonable target machines.

In object-oriented languages, method identification is not so simple. As we have seen in Section 11.2.9.4, in many cases a dispatch table must be consulted at run time to find the method to be invoked. The result of this consultation is again a routine value.

11.4.2.2 Routine calls—how to call it

Calling a routine is not just a transfer of control to the code of the routine; the **calling sequence** must also create the components of an activation record, as described in Section 11.3.2. Part of the calling sequence is performed at the routine call site and part is performed at the entry point of the called routine.

Before creating the components of the activation record, space for the activation record itself must be allocated. If a stack is used to store activation records, the space is allocated more or less automatically: the components are pushed onto the stack in the right order. If the allocation is explicit, the caller must allocate a chunk of memory of a suitable size, large enough to contain all components. This means that the required size must somehow be made available to the caller, for example as a run-time constant.

In the following discussion we will assume that activation records are allocated on a stack; this is the most usual situation. Heap allocation of activation records can be derived easily from the discussion below; where special measures are required, these are described.

A **stack** is a memory area onto which items can be pushed and from which items can be popped. Associated with a stack is a **stack pointer**, SP, which points to the "top" of the stack; the stack pointer resides in a dedicated register.

There is some confusion about which way stacks grow in memory. In abstract descriptions of stack machines, the usual convention is that pushing an item onto the stack raises the stack pointer; we used this convention in the pure stack machine of Section 7.5.2. On almost all real-world machines, however, pushing an item onto the stack *lowers* the numerical value of the stack pointer. This implies that the stack grows from high-numbered addresses to low-numbered addresses. Since we are considering code for real machines here, we will follow the hardware convention in this section.

There is an independent confusion about drawing stacks on paper: which way do they grow on paper? Although the word "stack" would suggest otherwise, stacks traditionally grow downwards in drawings, and we will follow that convention. Together the two conventions imply that memory addresses decrease from top to bottom in drawings, as was already illustrated in Figure 11.23.

The major advantage of using a stack for the activation records is that they do not have to be explicitly allocated and deallocated, thus saving considerably on allocation and deallocation times. Another advantage is that the working stack of the caller and the parameter area of the callee can be combined, which may save some time and space, as parameter values no longer have to be copied.

We will now discuss the individual components of the activation record, in the order in which they are usually created.

The parameter area and parameter transfer The parameters of the routine must be stored in or pushed to the location in which the callee expects them. The compiler must impose a rule for accomplishing this. An example of such a rule is: when parameters are passed on the stack, the last parameter is pushed first. Such a rule is suitable for languages that allow a variable number of parameters to be passed in a routine call and results in the **parameter area** shown in Figure 11.23 when the activation record is constructed on the stack. The compiler could also reserve a fixed number of registers for parameter passing and push the rest of the parameters onto the stack. Whatever scheme is chosen, it is essential that the caller stores the parameters in locations where the callee can access them.

If the routine returns a result, the parameter area may also contain a pointer to the location in which this result must be stored.

In some languages, a value can have a dynamic component, a component the size of which can only be determined at run time; the prime example is a dynamic array. For a parameter with a dynamic component, the dynamic component is allocated on the heap. The parameter part of the activation record then contains a pointer to the component, or a descriptor of known size with a pointer to the component.

Usually, the language manual specifies which parameter passing mechanism is to be used. The simplest and most common parameter passing mechanism is **call by value**: the rvalue of the actual parameter is used as the initial value of the corresponding formal parameter.

Many languages also support some form of output parameters, allowing a routine to change the values of some of its actual parameters by assigning to the corresponding formal parameters. There are two common mechanisms for this: **call by reference**, in which the change is effected immediately, and **call by result**, in which the change is effected upon return from the call. Both call by reference and call by result can be implemented by passing the lvalue of the actual parameter instead of its rvalue. The usual context condition here is that the actual parameter has an lvalue. In call by reference, an assignment to the formal parameter is implemented as an assignment to this lvalue. Call by result is implemented by allocating a local variable for the parameter, using this local variable throughout the routine, and assigning this local variable through the corresponding lvalue parameter in the return sequence.

Another common parameter passing mechanism is **call by value-result**, which is a combination of call by value and call by result. It is implemented using the scheme of call by result, with the addition that the local variable allocated for the parameter is initialized with the value of the actual parameter.

The administration part The **administration part** includes the frame pointer of the caller, which represents the dynamic link, and the return or continuation address. In languages that require so, it may also contain the frame pointer of the lexically enclosing routine, which represents the static link. During the initialization of the administration part, control is transferred to the callee, after saving the return address in the activation record of the callee or the continuation address in the activation record of the caller. Sometime during this phase, the old frame pointer is stored in the administration part and FP is set to indicate the new activation record.

The administration part also often contains space to save some machine registers. For example, sometimes machine registers are used for local variables and temporaries. When this is the case, these registers must be saved on routine call entry and restored on routine exit. There are two frequently used schemes for register saving and restoring: “caller saves” and “callee saves”. In the **callee-saves scheme**, the routine entry code of the callee saves all registers that may be corrupted by the callee’s code. In the **caller-saves scheme**, the routine call code contains code which saves the registers that the caller requires to be unharmed upon continuing after the call. In this scheme, the callee is free to use and corrupt all machine registers, since the caller has saved the ones it needs.

The caller-saves scheme usually requires fewer register saves and restores during run time, because it only has to save the registers active at the call, whereas in the callee-saves scheme all registers used by the routine need to be saved. On the other hand, the caller-saves scheme may require more instruction space, because every calling sequence needs to contain code to save and restore registers, whereas the callee-saves scheme only has code for this purpose at the routine entry and routine exit. Note that in the caller-saves scheme, the registers are saved in the caller’s activation record, whereas in the callee-saves scheme, they are saved in the callee’s activation record.

The local variable area Once control has been transferred to the callee, the callee can start building the **local variable area**, the part of the activation record in which local variables and compiler temporaries reside. The compiler can determine the size of this component from the sizes and alignment requirements of the local variables; it may even reorder variables with different alignment requirements to minimize the total size. The compiler also knows what temporary variables are required for the code of the routine, by keeping track of their number, size and alignment requirements during the code generation phase for this routine. This information is very dependent on the details of the code generation process, since optimizations may introduce or remove temporary variables. Space for the local variable area is allocated by decreasing the stack pointer by the proper amount.

As with parameters, one or more of the local variables may have a dynamic component; this dynamic component can be allocated on the heap, but for local variables there is another option, allocating it in the “dynamic allocation part” of the activation record, as explained below.

The working stack The local variable area is followed by the **working stack**, which is used for anonymous intermediate results from expressions. It may also be used for the bounds of for-statements, although it is more common to put these in temporaries.

Going through the code of a routine, the compiler can easily keep track of the size of the working stack and record its maximum size. At the end of the scan, the maximum size of the working stack for the routine is known; it can then be incorporated as a fixed-size block in the activation record.

The top of the working stack is indicated by the stack pointer as left by the allocation of the local variable area, and the working stack itself is the space below it. Machine instructions with push and pop properties can be used to access the working stack.

The dynamic allocation part If the target language allows activation records of dynamically extensible size, the activation record may also have a separate **dynamic allocation part** for local variables. These dynamic parts are then stored in the dynamic allocation part instead of on the heap. Since the dynamic allocation part is the only part the size of which cannot be determined statically, it has to come at the end of the activation record, as shown in Figure 11.45. In this set-up, the stack pointer points to the "top" of the dynamic allocation part.

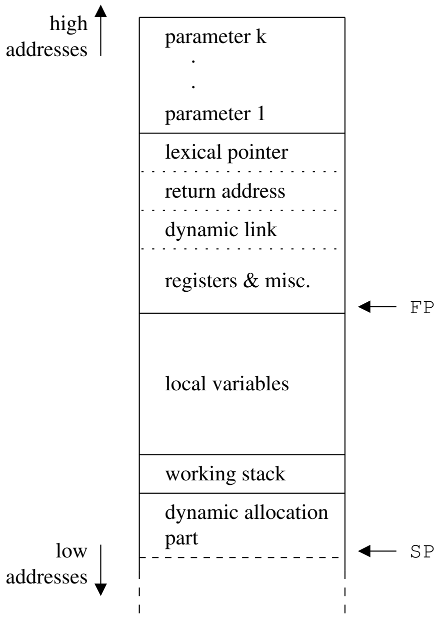


Fig. 11.45: An activation record with a dynamic allocation part

Some processors do not allow dynamic allocation parts, because they require the size of the activation record to be specified when it is created, by the compiler. For example, the SPARC processor has a save/restore mechanism for creating new activation records; the save part of the mechanism requires the size as a parameter. If the target language allows it, however, storing the dynamic part of local variables or parameters in the activation record has the advantage that it does not have to be deallocated explicitly on routine exit. Instead, its deallocation is implicit in the deallocation of the activation record. Also, allocation on the stack is much faster, since it only involves resetting the stack pointer.

A disadvantage is that the hardware stack pointer SP is now no longer available for manipulating the working stack. The remedy is to allocate the working stack somewhere in the activation record, and implement its stack pointer in software, for example in a normal register. It is not necessary to check this software pointer for stack overflow, since the size of the working stack is known in advance, and sufficient space can be guaranteed to be available.

Returning function results If the callee returns a value, the result can be made available to the caller in several ways. For a "simple" result type, such as an integer, real, or pointer, the compiler usually uses a machine register, called the **function result register**. The callee simply stores the result in this register, and the caller accesses this register to obtain the result. This register may serve other purposes as well, for example as a scratch register in the evaluation of expressions. For a "compound" result type, which means any result that does not fit naturally[1] into a register, the situation is more complicated. There are three reasonable solutions:

- If the compiler knows the size of the result, it can allocate a temporary variable in the data space of the caller, and pass its address as an extra parameter to the callee. The callee then stores the result through this address.
- Space for the result can be allocated dynamically by the callee, and a pointer to the allocated space is returned through the function result register.
- The result can be left on the working stack of the callee, in the dynamic allocation part of its activation record, or in one of its local variables, with a pointer to it in the function result register. Note that when activation records are allocated on the stack, the return sequence must make sure that the memory area in which the result resides is not overwritten. Also, the caller must then copy or use the result before it can use the stack again. When activation records are allocated on the heap, the caller must copy or use the result before releasing the activation record of the callee.

The calling and return sequences To summarize, the calling sequence consists of the following steps:

- Create an activation record.
- Evaluate parameters and store them in the activation record.
- Fill the administration part of the activation record. Entries include the frame pointer of the caller and the return address. They may also include the lexical pointer if required, probably some machine registers, and possibly also the old stack pointer.
- Transfer control to the callee.
- Make the frame pointer FP point to the activation record of the callee.

[1] Some data types, for example a record with just four character fields, fit unnaturally into a register. Most compilers will consider such result types "compound".

- Update the stack pointer SP, allowing enough space for the local variable part.

The dynamic allocation part, if present, is filled as the values of the local variables are computed.

The return sequence consists of the following steps:

- If the callee has a return value, store the result in the area designated for it.
- Restore the machine registers that were saved in the administration part.
- Update the stack pointer SP so that it indicates the frame pointer FP.
- Restore the frame pointer FP from the administration part.
- Transfer control to the caller, using the saved return address. Note that the administration part can still be accessed, because SP now indicates the activation record.
- Release the activation record of the callee.

Activation records and stack layout A typical layout of a stack with activation records is shown in Figure 11.46; the diagram depicts two activation records, that of a caller and that of its callee. We see that the two activation records overlap partially, since they share the parameter area. The caller evaluates the actual parameters of the call one by one, in last-to-first order, using its working stack. After evaluating a parameter, the result is left on the working stack, and the next one is evaluated on top of it. In this way, when all parameters have been evaluated, their values lie on the top of the caller's working stack. The working stack of the caller can now be used as the parameter area of the callee, resulting in overlap between the two activation records.

In this set-up, local variables are addressed through the frame pointer FP using negative offsets, parameters are addressed through FP using positive offsets, and the working area is addressed through the stack pointer SP. The position of SP in Figure 11.46 indicates an empty working stack.

If the implementation places activation records on the heap instead of on the stack, some of the techniques described above are not possible. Since activation records on the heap are separate entities, they cannot overlap and the working stack in one cannot serve as the parameter area in the other. Since records on the heap cannot be extended, activation records on the heap cannot have dynamic allocation parts.

Routine calling and parameter passing have received ample attention in the early literature; some pertinent papers are by Wichmann [298] and by Kowaltowski [161]. A very informative paper with strong opinions on the subject is by Steele [266].

Optimizations for routine invocations Several optimizations for routine invocations have already been mentioned above, the most important but at the same time most inconspicuous one being the allocation of the activation records on a stack rather than on the heap. Some others are the additional allocation of dynamic data on the stack, the precomputation of the size of the working area, and the overlapping actual and formal parameters.

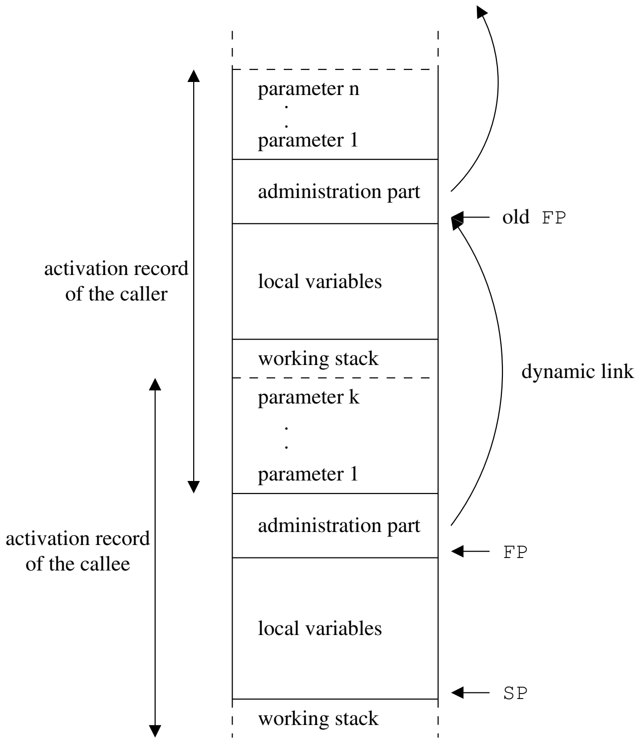


Fig. 11.46: Two activation records on a stack

There is another important optimization for routine invocations, which has more to do with flow of control than with activation records: tail call elimination. In its simplest form, the compiler replaces the code sequence

```
Call    routine
Return
```

with

```
Jump routine
```

This not only saves code and processor cycles, but it also unburdens the stack. When activation records come into play, the optimization becomes more complicated, but the concept remains the same.

A **tail call** is any routine call that is the last executed statement in a routine body. Routine bodies can contain more than one tail call; for example, in Figure 11.47 both gcd(b, a) and gcd(a-b, b) are tail calls. Tail calls are important for a combination of reasons: they occur frequently; they are easily detected; and, most importantly, they can be eliminated relatively easily, leading to sometimes considerable speed-up.

Although the frequency of tail calls in user programs of course depends on the subject, the programmer and the language, inspection of a few programs will quickly show that a sizeable number of routines end in a tail call. The code for quicksort,

```
PROCEDURE QuickSort(A: ArrayType);
  VAR Low, High: ArrayType;
BEGIN
   Split (A, Low, High);
  QuickSort(Low);
  QuickSort(High)
END
```

is a simple example, showing one tail call and two non-tail calls. The frequency of tail calls is even higher in generated code for functional and logic languages, as we will see in Chapters 12 and 13.

Tail calls are easy to detect in the control flow graph: they correspond to routine call nodes whose successor is a return node.

Much of the administration involved in a tail call is superfluous and can be eliminated. There are several reasons for this. First, there is no reason to return to the routine R that made the call (the *caller*) when the routine C that was called (the *callee*) has finished, since there is no additional code in the caller that needs to be executed. So there is no need for a return address when C exits, nor was there any need to store one when C was called: the transfer to C can be effected by a simple jump instruction. Second, the formal and local variables of the caller R are no longer required as soon as the parameters of the tail callee C have been computed, provided these parameters do not contain pointers to the formals or locals of R. And third, now that we have seen that the formals and locals can be abandoned and there is no need for a return address, most or all of the activation record of R is free just before the call and, with possibly some data juggling, can be used as the activation record of C.

```
int gcd(int a, int b) {
        if (b == 0) return a;
        if (b == 1) return 1;
        if (b > a) return gcd(b, a);
        return gcd(a−b, b);
}
```

Fig. 11.47: Recursive implementation of Greatest Common Divisor

When R and C are the same routine, we have a direct recursive tail call. We will discuss this situation first, using the Greatest Common Divisor (Figure 11.47)[2] as an example. The routine body contains two recursive tail calls. To eliminate the calls and reuse the activation record, we need to assign the new parameters to the

[2] There are much better algorithms for the GCD.

```
int gcd(int a, int b) {
 L_gcd:
        if (b == 0) return a;
        if (b == 1) return 1;
        if (b > a) {     /* return gcd(b, a); */
                int tmp_1 = b, tmp_2 = a;
                a = tmp_1, b = tmp_2;
                goto L_gcd;
        }
        {       /* return gcd(a−b, a); */
                int tmp_1 = a−b, tmp_2 = b;
                a = tmp_1, b = tmp_2;
                goto L_gcd;
        }
}
```

Fig. 11.48: Iterative implementation of GCD by tail call elimination

```
int gcd(int a, int b) {
 L_gcd:
        if (b == 0) return a;
        if (b == 1) return 1;
        if (b > a) {     /* return gcd(b, a); */
                int tmp_2 = a;
                a = b, b = tmp_2;
                goto L_gcd;
        }
        {       /* return gcd(a−b, a); */
                a = a−b;
                goto L_gcd;
        }
}
```

Fig. 11.49: Iterative implementation of GCD after basic bloc optimization

old ones, and replace the call by a jump. The first is most easily done by introducing temporary variables. The new parameters are assigned to the temporaries, and then the temporaries are assigned to the old parameters; this avoids problems with overlaps. The jump requires a label at the start of the routine body. This leads to the code of Figure 11.48. Basic block optimization can then be used to simplify the assignment sequences; this yields the code of Figure 11.49.

When *R* and *C* are not the same routine, there is still a simple way of eliminating the call to *C*: in-lining it. If that results in one or more directly recursive calls, *R* and *C* were mutually recursive, and we can eliminate the calls as explained above. If not, we can repeat the process, but all the caveats of in-lining (Section 7.3.3) apply. This approach fails if not all source code is available, for example, when calling a separately compiled (library) function. Most mutually recursive functions, however, occur in the same module.

When generating assembly code, eliminating the call is even easier, since the start of the code of *C* is available. The parameter values of *C* are written over the activation record of *R*, using temporary variables as needed; the activation record then looks like a proper activation record for *C*, and a simple jump to the start of the code of *C* suffices for the transfer of control.

Tail recursion removal was introduced in 1960 by McCarthy [186], to assist in the translation of LISP. A good description of tail recursion removal with examples and an evaluation is given by Bailey and Weston [25]. Formal foundations of tail recursion elimination are presented by Clements and Felleisen [63].

11.4.2.3 Non-local gotos

The flow-of-control aspects of non-local goto statements have already been covered in Section 11.3.6.4. We will consider here the data handling aspects.

The main data issue of the implementation of a non-local goto statement is that the registers that were saved in the intervening activation records must be restored when these activation records are destroyed. In the absence of special measures there is no proper way to do this, because there is no way to find out which registers to restore from a given activation record. A special measure that can be taken to prevent this problem is to record in each activation record which saved registers it contains. Restoring the registers then becomes trivial, at the cost of some extra book-keeping at each routine call. For some target machines, it may be feasible to always, in each activation record, save and restore the same set of registers, so that it is not required to record which registers are saved. Another option is not to use register variables in routines that can be the target of a non-local goto. These routines must then also save all registers on routine entry, and restore them on routine exit. This technique allows the implementation not to restore registers from intervening activation records.

We will now turn to a form of control flow in which the non-local goto plays a significant role.

11.4.3 Run-time error handling

There are many ways a program can get into trouble. Some examples of problems are: integer overflow, an array bound error, memory exhaustion, and power failure. Depending on circumstances, the generated code, the run-time system, the operating system, or the hardware may detect the error, and then some level of the system will probably cause some error message to be displayed and the program to be terminated. Or the error may go unnoticed. Since this is often not the desired situation, some languages allow the programmer to specify which errors must be caught and also to specify the actions to be performed upon the occurrence of these errors or exceptional situations. So, handling of a run-time error consists of two parts: the error must be detected and the actions for the error must be performed. Both have their problems: detecting the error may be difficult and some of the required actions are very inconvenient.

11.4.3.1 Detecting a run-time error

A run-time error can be caused either by the program itself or by external circumstances. Detecting a run-time error caused by external circumstances is very system-specific. For example, a memory failure—the malfunction of a memory section—may or may not be made available to the program. It is up to the run-time system to make errors caused by external circumstances available if it can.

Often, the operating system makes exceptional external circumstances available to the run-time system by causing an interrupt. For example, on the UNIX operating system, a user can interrupt a running program by typing a control-C or similar keystroke. The default action on an interrupt is to terminate the program, but UNIX includes a mechanism for making the interrupt available to the program. The program can then detect that an interrupt has occurred, and take appropriate action.

Examples of run-time errors caused by the program itself are: integer overflow, array bound errors, division by zero, and dereferencing a null pointer. Some of these errors cause the operating system to give an interrupt, but often the compiler must generate checks to detect these run-time errors. Several error conditions can be checked for by examining certain bits in the processor status word. For example, there is usually an overflow bit, indicating that the last instruction performed caused an overflow. Similarly, a library or the program itself may also detect run-time errors, for example file access errors, or program-specific problems such as illegal user input, internal errors, or security violations.

Most processors lack special facilities for higher-level error conditions, for example array or range bound errors. If such errors are to be detected, the compiler must generate code to perform a check, testing that a certain integer falls within a range. Such a check basically looks as follows:

```
IF (val < lowerbound) OR (val > upperbound) THEN
    THROW range_error;
END IF;
```

where THROW is a basic statement to signal an error. These checks can be quite expensive, which is why many compilers have a switch that disables the generation of these checks.

11.4.3.2 Handling a run-time error

When a programming language does not have facilities to allow the user to deal with run-time errors, the story ends here. The run-time system can make sure that it produces an intelligible error message, and then may terminate the program, producing some kind of memory image, usually called a “memory dump”, which can then be examined using a post-mortem debugger. Many languages, however, include a mechanism for dealing with run-time errors. We will now discuss two such mechanisms, signal routines and exceptions.

Signal handlers A simple mechanism for dealing with run-time errors is the **signal statement**. A signal statement binds a specific class of error conditions to a specific user-defined routine, the **signal handler**. Whenever a run-time error occurs, the corresponding signal handler is called. The signal handler might then close open files, free system resources, print a message, and terminate the program. Alternatively, it could deal in some other way with the error, and just return. In the latter case, the program will resume just after the point where the error occurred. To implement the signal statement, the run-time system maintains a program-wide list of (error condition class, signal handler) pairs, so it can call the corresponding signal handler when a run-time error occurs.

Sometimes it is convenient to have the program continue at a point different from where the run-time error occurred. For example, a program could be in a main loop processing commands. A run-time error in one of the commands should then cause the program to print an error message and continue with the main loop. The programmer can implement this by using a non-local goto statement in the signal handler to transfer to the main loop of the program. In a non-nesting language like C, the setjmp/longjmp mechanism can be used, as shown in Figure 11.50.

```
#include <setjmp.h>

jmp_buf jmpbuf;          /* type defined in setjmp.h */

void handler(int signo) {
    printf ("ERROR, signo = %d\n", signo);
    longjmp(jmpbuf, 1);
}

int main(void) {
    signal(6, handler);         /* install the handler ... */
    signal(12, handler);        /* ... for some signals */
    if (setjmp(jmpbuf) == 0) {
        /* setting up the label for longjmp() lands here */
        /* normal code: */
        ...
    } else {
        /* returning from a call of longjmp() lands here */
        /* exception code: */
        ...
    }
}
```

Fig. 11.50: An example program using setjmp/longjmp in a signal handler

Exception handlers A more flexible feature, available in many modern programming languages, is the facility to specify "exception handlers". An **exception handler** specifies a particular error condition and a set of statements that will be executed should this error condition occur. This allows an exception handler to access the local variables of the block or routine in which the error condition occurred. The statements in the exception handler replace the rest of the statements in the block or routine, should the corresponding error condition, or "exception", occur.

Usually, each block of statements or each routine can have its own exception handler. It can even have several exception handlers, for different error conditions.

When an exception E occurs, the chain of activation records is followed, releasing activation records on the way as in the implementation of a non-local goto, until one is found that has a handler for the exception E. This handler is then executed, and execution resumes at the end of the block or routine that the handler is associated with—unless the exception handler terminates the program or causes another exception.

For each exception handler, the compiler generates the code corresponding to the statements in the exception handler, terminating it with a jump to the end of the block associated with the handler, and labels this code with a unique handler label. Also, for each block or routine with exception handlers it generates a table of (exception, handler label) tuples, with one entry for each exception handler.

In addition, the administration part of each activation record must contain a pointer to the table of exception handlers currently installed for its routine. If the exception handlers are associated with a routine, this is a constant; if they are associated with a block, the pointer must be updated as blocks with exception handlers are entered and left. Now, when an exception E occurs, the chain of activation records is examined as follows, most recent activation record first:

1. The pointer to the table of exception handlers is extracted from the currently examined activation record.
2. The table referenced is searched for a handler for exception E. If such a handler H is found, step 3 is performed. If not, step 1 is performed on the parent activation record (the one of the caller). If there are no more activation records, the program is terminated, possibly after printing a message.
3. A non-local goto to the exception handler H is performed. This automatically releases all examined activation records that turned out not to have a handler for exception E. Note that the generated code for the exception handler takes care of the continuation of the program execution, once the handler is finished.

The discussion above leads us to the important observation that routines with exception handlers can be the target of a non-local goto. So, depending on the implementation of non-local gotos, the compiler may or may not keep variables of routines with exception handlers in registers, in accordance with the preceding discussion on non-local gotos.

A disadvantage of the method described above is that, depending on the implementation of the non-local goto, activation records may be examined twice, once to find a handler, and once to perform the non-local goto. An optimization to this approach is to restore information from an examined activation record when it turns out not to have a handler for the exception, and to then discard the activation record.

The exception handler mechanism is more flexible than the signal handler mechanism. Its increased flexibility comes, however, at the cost of a slight increase in the number of instructions required to perform a routine call or block entrance; these instructions are necessary for the construction of the exception handler table. Other exception handler implementations are feasible, of which we will discuss one example.

An alternative implementation replaces the (exception, handler label) pairs mentioned above by a program-wide list of tuples (exception, handler label, start address, end address). Here "start address" indicates the beginning of the block protected by the handler, and "end address" indicates the end of that block. When an exception occurs, the run-time system uses this list of tuples to determine which handler should be invoked, as follows. First, the value of the program counter at the time the exception occurred is compared with the start and end addresses of blocks protected by a handler for the exception at hand. If it falls within a start address–end address range, a handler is found, and a non-local goto is performed, as discussed above. If not, the return address is extracted from the activation record at hand. This return address represents the position in the code where the caller currently is. This position may again be protected by a handler, so, again, it is compared with the start and end addresses of blocks protected by a handler for the exception at hand. Again, as above, if a handler is found, a non-local goto is performed. If not, the return address of the caller is extracted, etc. Ultimately, either a return address (and an activation record) with a handler is found, or execution terminates.

This scheme is more efficient as long as no exceptions occur. However, searching for a handler may be more expensive, depending on the exact implementation of the exception handler table.

11.5 Code generation for modules

Modules (also called "packages") are syntactic structures in which a number of related items are grouped together. They often restrict access to their contents by providing an explicit interface, which is then the only means of accessing the module. The related items in a module could for example be variables and routines affecting these variables. Modules are somewhat similar to objects, but there are also considerable differences. In some respects they are simpler: they usually cannot be created dynamically, and lack all the object-oriented features discussed in Section 11.2.9. On the other hand, modules are compilation units and it must be possible to compile them separately; second, modules often require an explicit initialization.

Regarding code generation, modules introduce two problems to the compiler writer:

- The target language usually has one, flat, name space. Therefore, the compiler must make sure that if two different modules export an item of the same name,

they have a different name in the generated code.
- The compiler must generate code to perform the module initialization; a complication here is that at module initialization modules may use items from other modules, so these other modules must be initialized first.

Roadmap

11.5.1 Name generation

Usually, the rules that the characters in a name must obey are more strict in the source language than in the target language. Often, there is a character *c* that is allowed in names in the target language but not in names in the source language; examples are the . (dot) and the $ sign. If so, this feature can be used to create unique names for items in a module: simply concatenate the item name to the module name, using *c* as a separator. Note that this assumes that module names are unique, which they usually are.

If there is no such character, there may be some other rules in the source language that can be exploited in the compiler to produce a unique name. If all else fails, the compilation process could have a phase that analyzes the complete program and does name generation.

11.5.2 Module initialization

Most source languages allow modules to have explicit **initialization code**, for example for global variables. Even if this is not the case, a language might require implicit initialization of these variables. Note that a module, and more specifically the initialization code of a module, could use items from other modules. This means that those modules should be initialized earlier. This can be accomplished by having the initialization code of each module call the initialization code of all the modules that it uses. Then, the whole initialization phase can be started by calling the initialization phase of the module that contains the main program.

When adopting this solution, there are two issues one should be aware of: avoiding multiple initializations and detecting circular dependencies.

11.5.2.1 Avoiding multiple initializations

If module *A* uses module *B* and module *C*, and module *B* also uses module *C*, the initialization code of module *A* calls that of module *B* and that of module *C*. The initialization code of module *B* also calls that of module *C*, so *C*'s initialization will get called twice. This can be prevented by having a module-specific variable *ThisModuleHasBeenInitialized* in each module, which is set to true once its initialization code is called. The initialization of a module then becomes:

```
if not ThisModuleHasBeenInitialized:
    ThisModuleHasBeenInitialized ← True;
    –– call initialization of the modules used by this module
    –– code for this module's own initializations
```

11.5.2.2 Detecting circular dependencies

Circular dependencies between module *specifications* are usually detected by the compiler. When compiling a module specification *A* that imports a module specification *B*, the compiler usually demands that module specification *B* is present for examination or has already been compiled. If module specification *B* then requires module specification *A*, the compiler can easily detect that it already was compiling *A*, and deal with this according to what the language manual prescribes (which is probably to disallow this).

For module *implementations* the situation is different. When compiling a module implementation *A*, which uses module specification *B*, the compiler can see what module specifications are used by the module specification for *B*, but it cannot know what module specifications are used by *B*'s implementation, since the latter need not even be available yet. Now suppose the present or a future *B* implementation uses *A*'s specification. Then we have a circular dependency, which is not detected at compile time, because when compiling *B*'s implementation, the compiler only reads *A*'s specification, which in our example does not use *B* at all.

One way to detect circular dependencies is to postpone this check to run time, during module initialization. Each module then has a module-wide Boolean variable *ThisModuleIsBeingInitialized*, and the initialization code then becomes:

```
if ThisModuleIsBeingInitialized:
    –– circular dependency; deal with it
if not ThisModuleIsInitialized:
    ThisModuleIsInitialized ← True;
    ThisModuleIsBeingInitialized ← True;
    –– call initialization of the modules used by this module
    ThisModuleIsBeingInitialized ← False;
    –– code for this module's own initializations
```

A disadvantage of this approach is that the error does not become apparent until the compiled program is run. A more elegant solution to this problem is to have the compiler produce a list of the modules that each module uses. A separate compilation phase, which is invoked after all module implementations have been compiled, may then read these lists and detect circular dependencies.

Such a phase could also be used to determine a global initialization order, by imposing a relation $<$ on the modules, such that $A < B$ means that the initialization of A must be called before the initialization of B. In the absence of circular dependencies, a topological sort then gives the global initialization order. The presence of such a phase allows the generation of an initialization routine, to be called at program startup, which calls the module initializations in the right order. The module initializations themselves then only perform the initialization of the module itself. It is, however, not always convenient or practical to have such a phase.

11.5.3 Code generation for generics

Many languages offer generic units. A **generic unit** is a recipe for generating actual versions of the unit; it is parameterized with one or more generic parameters, usually types. Generic units are usually limited to routines, modules, and object classes. A generic unit must be instantiated by supplying the actual parameters to produce an instance of the generic unit, which can then be used like any other unit in the program. Section 2.12.3 presented the classic example of a generic list.

When generating code for a generic unit, we are faced with the question of how to implement a generic parameter, especially when it is a type. There are two fundamental ways to deal with this problem. The first way is to not produce code for the generic unit itself, but rather to produce code for every instantiation. This can be viewed as expanding the generic unit as if it were a macro, as already suggested in Section 2.12.3 in the context of macro processing. Alternatively, the compiler writer can design run-time representations of the generic parameters whatever they are, and pass them as normal parameters during run time. Given the unusual nature of generic parameters, they have unusual representations, called “dope vectors”.

11.5.3.1 Instantiation through expansion

In **instantiation through expansion**, no code is generated for the generic unit, which remains available as an AST only. Suppose we have a generic unit G with a generic parameter type T, and suppose the generic unit is instantiated with actual parameter type *tp*. The instantiation then consists of the following steps:

- Create a copy of the AST of the generic unit G.
- In the copy of the AST, replace every occurrence of identifier T which is identified as the generic parameter type T by the type indicated by the identifier *tp* at the instantiation point.
- Process the resulting AST as if it were an ordinary AST, resulting from an ordinary unit.

An important issue that must be treated with care is name generation, also discussed in Section 11.5.1. Usually, the target language has a single, flat, name space. Therefore, the compiler must generate a unique name for each instantiation. We cannot just use the generic unit's name—which is supposed to be unique—since a generic unit may be instantiated more than once. However, the instantiation probably also has a name, and the unit in which the instantiation takes place also has a name. A combination of all these names allows the compiler to produce a unique name for the items in the instantiated unit.

Instantiation through expansion is relatively easy to implement and there is no run-time overhead. It may, however, increase the size of the generated code significantly, especially when a generic unit contains instantiations of its own. Of course, the compiler could keep track of which instantiations it has already performed and reuse those when possible, but this will not always help, because all instantiations may be with different parameters.

11.5.3.2 Instantiation through dope vectors

In **instantiation through dope vectors**, the compiler actually produces code for the generic unit from the AST. The generated code will have to utilize run-time descriptors for the instantiation parameters. The run-time descriptor of an instantiation parameter must contain enough information to support all possible usages of a generic parameter of its kind: for a constant, it should contain its value; for a routine, it should contain its address and a lexical pointer, if applicable; and for a type, it should contain a so-called **dope vector**.

If a generic parameter is a type *tp*, the compiler will have to generate code to allocate, initialize, and deallocate variables of type *tp*, and to perform operations that involve *tp*, for example assignment, comparison, copying, and possibly arithmetic operations of various lengths. Since these are operations that very much depend on the nature of the type *tp*, it is convenient for the compiler to generate calls to routines which perform these operations. The addresses of these routines can then be found in the run-time descriptor of the instantiation parameter, the dope vector. This makes a dope vector very similar to a method table with entries for a constructor, a destructor, a copy routine, etc.

So, the dope vector has an entry for each operation that is allowed on a value or variable of a generic parameter type, including allocation, deallocation, and copying. Each entry indicates a routine implementing the specific operation, as depicted in Figure 11.51.

The main advantage of the use of dope vectors is that the executable code of the generic unit is shared by all its instantiations. A disadvantage is the added overhead: often, dynamic allocation schemes must be used for types for which simple static allocation would be sufficient. Also, there is the routine call overhead for simple operations such as assignment and comparison. Note that this overhead cannot be eliminated by in-lining the call, since the routine to be called is not known until run time.

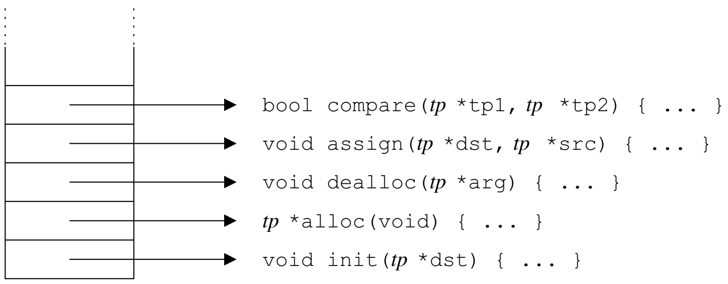

bool compare(*tp* *tp1, *tp* *tp2) { ... }
void assign(*tp* *dst, *tp* *src) { ... }
void dealloc(*tp* *arg) { ... }
tp *alloc(void) { ... }
void init(*tp* *dst) { ... }

Fig. 11.51: A dope vector for generic type *tp*

11.6 Conclusion

This brings us to the end of our discussion of the strongly interrelated issues of intermediate code generation and code generation for run-time systems, for imperative and object-oriented languages. We have seen that the most important run-time data structure is the activation record: it allows both data and the flow of control to move from routine to routine. Code for its manipulation can be generated in-line or be contained in run-time library routines. The design of other run-time data structures, which derive from source language type constructors, is fairly straightforward. We have seen that all source language flow of control is expressed by simple unconditional and conditional jumps in the intermediate code; the generation of code for expressions has already been covered in Chapters 7 and 9. Special attention was paid to the selection of the proper method to call in an object-oriented method invocation; dispatch tables of varying complexity are used to implement specific object-oriented features. Another area in which intermediate code and run-time system are interwoven is exception handling. Finally we have discussed issues in the implementation of modules and of generics.

Summary

Summary—Context handling

- Static correctness checking during context handling consists mostly of two issues: identification and type checking.
- Identification matches an applied occurrence of an identifier or operator with its defining occurrence, and with a symbol table entry, in which the information about the item is collected.
- A scope stack allows fast and easy manipulation of scope information sets.
- Some languages allow identifiers or operators to have several different definitions through overloading. Overload resolution is the process of reducing the set of

possible definitions of an operator or identifier to a single one.
- Type checking is the process that determines which type combinations in the source program are legal according to the rules of the source language. It does so by determining which types are equivalent, and deciding which coercions are allowed in which context.
- Each node in the AST of an expression represents either an lvalue (a location) or an rvalue (a value). These attributes propagate bottom-up, according to language-dependent, but usually intuitive, rules.

Summary—Data representation and routines

- Characters, integers, floating point numbers, enumerations, and pointers all have an obvious corresponding data type in virtually all target languages.
- The target representation of a record type consists of the consecutive target representations of its members, possibly separated by gaps to obey alignment requirements of the target machine.
- The target representation of an array type consists of the consecutive target representations of its elements, either in row-major or column-major order. In addition, a run-time array descriptor is required for dynamic arrays.
- The run-time representation of an object consists of two parts: (1) a pointer to a table for the present implementation of the object type, and (2) a record holding the object fields. The table is a dispatch table for the methods, and possibly an offset table for the fields.
- Inheritance and polymorphism make it necessary to enumerate object methods, and use the enumeration index of a method to obtain its address at run time from the dispatch table. Dependent multiple inheritance requires a similar technique to find the offset of a field within an object representation, using the field offset table.
- An activation record holds the data pertinent to the invocation of a routine or object method. It includes the parameters, an administration part, local variables, and a working stack.
- If routines are activated in a strictly last-in-first-out order, a stack can be used to store the activation records.
- An activation record is accessed through a frame pointer, which usually resides in a dedicated machine register.
- A routine is a piece of code, reachable through its code address. When a routine is called, an activation record is created for it and flow of control is transferred to its first instruction.
- The above actually is a subroutine. Other kinds of routines are iterators and coroutines.
- An iterator yields a succession of values.
- A coroutine can suspend itself, and does so by transferring control to another coroutine, which then resumes at the point where it was last suspended.

- Routines can be global or nested. A global routine can only access global entities and its own local entities. A nested routine can access those, and can in addition access entities in lexically intervening routines.
- Operations on routines include declaring a routine (specifying its name, parameter types, and return type), defining a routine (declaring it and supplying its code), calling a routine, passing a routine as parameter, returning a routine as a value, jumping out of a routine (a non-local goto), and partial parameterization (creating a new routine with some of the parameters of the original routine already supplied).
- Nested routines require access to lexically intervening routines. This can be implemented by means of a lexical pointer, which is a copy of the frame pointer of the lexically enclosing scope.
- Passing a nested routine as a parameter or returning it as a result requires a two-pointer value: a lexical pointer and a routine address. Returning a nested routine as a result may violate pointer scope rules.
- A closure is a representation of a partially parameterized routine, with space for all parameters plus a mask indicating which parameters have been filled in.
- A nested routine can be lifted out of its parent by isolating the activation record of the parent routine as a data structure and passing it to the lifted routine. This allows all the access the nested routine had.
- Any place a lifted routine is called, the activation record of its parent must be passed to it. In particular, if a lifted routine is passed around, the activation record of its parent must be passed along with it; they form a (routine address, activation record address) pair.

Summary—Code generation

- The two main mechanisms for local flow of control are selection (if- and case statements) and repetition (for- and while statements). Code is generated for them by mapping them to lower-level target language constructs, such as conditional jumps and assignments.
- Code for if-statements is trivial, using true and false labels. The jump can be integrated into the Boolean expression by passing the true and false labels to the routine that generates code for it.
- Case statements can be implemented as linear lists of tests, jump tables, balanced binary trees, and others.
- Code for the while statements is similar to that for the if-statement, including the incorporation of the jump into the Boolean expression.
- The for-statement poses two problems: in a naive implementation the controlled variable may overflow where it should not, and for non-unit step size, the upper bound may be overstepped. Both problems are remedied by a careful implementation; the resulting code is surprisingly complex.

- The code for repetition statements can be optimized by “unrolling the loop”, generating the code for the loop body multiple times.
- The translation of a routine call or method invocation consists of two steps: identifying the routine or method called, and performing the call.
- The routine to be called can be a named routine, a routine variable or a method. The first two identify themselves, the third is looked up in the dispatch table.
- A routine is called in three steps: an activation record is allocated, it is partly filled with information, and control is transferred to the start of the code of the routine; filling the activation record may then continue. Many detailed decisions are required to design a complete routine call protocol.
- The information stored in the activation record may include parameters, return or continuation information, a lexical pointer, a dynamic link (old frame pointer), dumped register values, local variables, working stack, dynamic allocation part, etc.
- Storing of register values may be done by the caller (“caller saves”) or by the callee (“callee saves”), with subtly different effects. In both cases it is the register values of the caller that get saved.
- A dynamic allocation part may be present to hold the dynamic parts of local variables and perhaps parameters.
- In most languages activation records can be allocated on the stack. If routines can be returned as values (functional languages), or if more than one thread is present (coroutines) this is not possible and heap allocation must be used.
- A return value can be delivered in a function result register, on the top of the working stack, or under a pointer passed as an input parameter.
- An important optimization on routine calls is tail call elimination, which can often optimize away the entire routine call.
- When generating assembly code, the tail call inside a routine R can be optimized into a stack adjustment plus jump to the new routine, C.
- In C we can only jump to the start of the current routine. This allows us to optimize the important case of directly tail-recursive functions by overwriting the parameters with the values of the tail call and jumping back to the start of function R.
- While performing a non-local goto, the registers stored in the activation records that are removed must be restored. Special code is required for this.
- Run-time error handling (exception handling) consists of two parts, detecting the error and processing the error.
- Detection of run-time errors depends in complicated ways on language properties, the operating system, the run-time system, and compiler-generated checks. Two different mechanisms for dealing with run-time errors are: signal handlers, and exception handlers.
- A signal statement binds a specific class of error conditions to a specific user-defined routine, the signal handler. Whenever a run-time error occurs, the corresponding signal handler is called.
- To implement the signal statement, the run-time system maintains a program-wide list of (error condition class, signal handler) pairs, so it can call the cor-

responding signal handler when a run-time error occurs. Signal handlers often perform a non-local goto.
- An exception handler specifies a particular error condition and a set of statements that will be executed should this error condition occur. The statements in the exception handler replace the rest of the statements in the block or routine, should the corresponding error condition, or exception, occur.
- When an exception E occurs, the chain of activation records is followed, releasing activation records as in the implementation of a non-local goto, until one is found that has a handler for the exception E. This handler is then executed, and execution resumes at the end of the block or routine that the handler is associated with.
- The administration part of each activation record contains a pointer to the table of exception handlers currently installed for its routine. If the exception handlers are associated with a block, the pointer must be updated as blocks with exception handlers are entered and left.
- The presence of modules in a source language forces the compiler to generate unique names, because the target language usually has a flat name space.
- The initialization phase must make sure that modules are initialized in the proper order, obeying module dependencies.
- The two common implementations for instantiation of generic units are expansion, in which the compiler processes the generic unit at the instantiation point as if it were an ordinary unit, and dope vectors, in which the compiler generates run-time descriptors, the so-called dope vectors, for the generic parameters of the generic unit.

Further reading

For many years, the only programming languages in existence were imperative ones. Examples of imperative programming languages are: FORTRAN, Algol 60, Pascal, Algol 68, Modula-2, C, Ada, and many, many more. Several of these languages are still quite popular. Consequently, all compiler construction books discuss the implementation of imperative programming languages.

The first object-oriented language was Simula. It introduced classes, inheritance, and polymorphism. It was originally designed specifically for simulation problems (hence the name), but was later developed into a complete programming language. Other better-known object-oriented languages are Smalltalk, C++, and Java. There seem to be no books that are specifically about the compilation of object-oriented languages, although both Appel [18] and Wilhelm, Seidl and Hack [113, 300–302] pay considerable attention to the subject. For the more theoretically inclined, Duran *et al.* [91] describe an algebraic formalism for the derivation of compilers for object-oriented languages.

Many advances in the compilation of imperative and object-oriented languages are reported in the conference proceedings of the *ACM SIGPLAN Conference on*

Programming Language Design and Implementation - PLDI, the *Conference on Object-Oriented Programming Systems, Languages and Applications - OOPSLA*, the *IEEE International Conference on Computer Languages - ICCL*, and the *European Conference on Object-Oriented Programming - ECOOP*.

Exercises

11.1. (▷www) Why is the result of the expression `EXPR >= 0` trivial, when `EXPR` is an expression of type unsigned integer?

11.2. (▷794) In Section 11.1.1, Figure 11.4, we first follow the hash table and then split according to the name space desired. Perhaps a more intuitive way of implementing name spaces is to have a separate hash table for each name space, and pass a pointer to it as a parameter to the routine *Identify()*. A call *Identify (NameSpace, Name)* would then return a pointer to the proper *DeclarationInfo*. Discuss the pros and cons of this idea.

11.3. (▷www) The following declarations are given for a language that uses name equivalence:

```
A, B: array [1..10] of int;
C: array [1..10] of int;
D: array [1..10] of int;
```

Explain which of these four variables have the same type and which have different types.

11.4. Explain the transformations the type of a routine undergoes in partial parameterization.

11.5. Determine which nodes carry lvalues, which carry rvalues and where dereference operators have to be added, in the AST for the expression `a[a[1]] := s[0].sel`; assume that `a` is an integer array variable and `s` is a variable array of structures.

11.6. (▷794) Design rules for the lvalue/rvalue checking of the C conditional expression *condition*?$expr_1$:$expr_2$. Compare your rules with the behavior of the GNU C compiler, with and without `–ansi` and `–pedantic` flags.

11.7. (▷794) One of the scope rules in Section 11.2.3.2 says "The scope of a value is the smallest scope of any pointer it contains or infinite if it does not contain pointers." What if the value contains pointers with incommensurable scopes?

11.8. (▷794) Why are the following expression modifications incorrect for IEEE 754 floating-point computations:

1. `a+(b+c)` $\rightarrow$ `(a+b)+c`
2. `a*(b*c)` $\rightarrow$ `(a*b)*c`

3. v == v → `true`

11.9. (▷www) In C, why is it that assignment (=) *is* defined for structs and equality (==) is not? (Hint: this *is* a compiler construction question.)

11.10. (▷www) Given the C structure

```
struct example2 {
    int member1;
    double member2;
    int member3;
}
```

where an int is 4 bytes and a double is 8 bytes, and an int must be 4-byte aligned, and a double must be 8-byte aligned.
(a) What is the size and alignment of struct example2?
(b) What happens if fields member2 and member3 are interchanged?
(c) Could an optimizing compiler consider reorganizing record fields to achieve minimal size?

11.11. (▷www) Section 11.2.6 suggests that the compiler can compute the offset of *zeroth_element* from the first element of the array itself. Let us call this offset *zeroth_offset*. Give the formula for *zeroth_offset*(A) and the formula for the address of A$[i_1, \ldots, i_n]$ using *base*(A) and *zeroth_offset*(A) instead of *zeroth_element*(A).

11.12. (▷www) Given a three-dimensional array and a three-deep nested set of arrays (i.e. an array of arrays of arrays), with all arrays bounds-checked and starting at element 0, compute for both how many multiplications, additions, comparisons, and memory dereferences are needed. What if the arrays do not start at element 0?

11.13. (▷www) Show that it is possible to implement a bounds check of a Java array with only one comparison.

11.14. (▷www) Refer to Section 11.2.7 on set types. Discuss the implementation of set union and set intersection when implementing a set as a linked list.

11.15. (▷www) Consider the following classes (presented in a Java-like syntax):

```
abstract class Shape {
    boolean IsShape() { return true; }
    boolean IsRectangle() { return false; }
    boolean IsSquare() { return false; }
    abstract double SurfaceArea();
}

class Rectangle extends Shape {
    double SurfaceArea() {
        ...
    }
    boolean IsRectangle() { return true; }
}
```

```
class  Square extends Rectangle {
    boolean IsSquare() {  return  true ;  }
}
```

Give the method tables of the classes Rectangle and Square.

11.16. (▷794) When a language supports polymorphism, as described in Section 11.2.9.3, it is sometimes necessary to examine the actual type of the object at hand. The example of Exercise 11.15 has methods like IsSquare that enable the user to do this. Some languages have a built-in operator for this. For instance, Java has the instanceof operator. The expression A instanceof C is a Boolean expression yielding true if A indicates an object that is an instance of class C. Design an implementation for this operator.

11.17. (▷794) Refer to Figures 11.18 and 11.19 concerning multiple inheritance. Given an object e of class E, give the compiled code for the calls e.m1(), e.m3(), and e.m4().

11.18. (▷www) Refer to Figures 11.21 and 11.22 concerning dependent multiple inheritance. Suppose method m5 in class E is defined as

```
void m5() {
    e1 = d1 + a1;
}
```

where all fields have type int. Give the compiled C code for m5.

11.19. (▷795) Explain why the “caller saves” scheme usually requires fewer register saves and restores at run time than “callee saves”.

11.20. Given the GNU C routine:

```
void A(int A_par) {
    void B(void) {
        printf("B called, A_par = %d\n", A_par);
    }
    void C(int i) {
        if (i == 0) B(); else C(i–1);
    }
    C(5);
}
```

(a) Draw the stack that results from the call A(3).
(b) How does the final call of B() access the parameter A_par of A?
(c) Repeat parts (a) and (b) for the routine

```
void A(int A_par) {
    void C(int i) {
        void B(void) {
            printf("B called, A_par = %d\n", A_par);
        }
        if (i == 0) B(); else C(i–1);
    }
    C(5);
}
```

11.21. (▷www) (F.E.J. Kruseman Aretz, mid-1960s) Write a program that reads an array of integers terminated by a zero, and prints its middle element, *without* using an array, linked list, or otherwise allocated data. Hint: Read the elements of the array recursively, on each level of recursion prepare a routine element(int i), which returns the i-th element, and pass this routine to the next level.

11.22. (▷www) Why is it that the closure must be *copied* to the activation record in the invocation of a routine represented by a closure? If the closure is allocated in an extensible array on the heap, it could itself be turned into an activation record, could it not?

11.23. Study the coroutine mechanism of Simula 67 [41], and design intermediate code sequences for the new, detach and resume commands.

11.24. What code should be generated for the Boolean assignment b := x > y, in which b is a Boolean variable with the representation true = 1 and false = 0, and x > y translates to Compare_greater x, y, which leaves the result in the *condition_register*?

11.25. (▷www) The discussion of case statements in Section 11.4.1.2 mentions using a balanced tree for the case labels as a possible implementation for the case statement. Why does the tree have to be balanced?

11.26. (▷www) Discuss the translation of a case statement by means of a hash table.

11.27. (▷www) A repeat statement allows the programmer to execute a sequence of statements an indeterminate number of times, until a *Boolean_expression* is fulfilled. Give a translation of the statement

REPEAT *statement_sequence* UNTIL *Boolean_expression* END REPEAT;

to intermediate code. Note that the *statement_sequence* is executed at least once.

11.28. (▷www) The C language has a break and a continue statement. The break statement terminates the closest enclosing loop, and the continue statement proceeds with the next iteration. In fact, these are both jumps. In the code generation schemes of Section 11.4.1.3, where would these go to? In other words, where should the compiler place the break_label, and where the continue_label?

11.29. (▷www) Why is the C while loop of Figure 11.43 not exactly equivalent to the for-loop of Figure 11.42? Hint: consider the effect of a continue statement inside the body.

11.30. (▷795) Refer to Figure 11.41. What goes wrong if one generates

```
(1)    i := i + tmp_step_size;
(2)    IF tmp_loop_count = 0 THEN GOTO end_label;
(3)    GOTO loop_label;
end_label:
```

in which statements (1) and (2) are interchanged with respect to the original?

11.31. (▷www) The optimization explained in the last subsection of Section 11.4.1.3 replaces a loop with a step size of 1 by a loop with a step size of 4, thus apparently introducing all the complexity of the code of Figure 11.41. Devise a simpler code scheme for such loops. Make sure it is still overflow-resistant.

11.32. (▷795) In Section 11.4.2.2 we claim that pushing the last parameter first onto a stack is a technique suitable for languages that allow a variable number of parameters to be passed in a routine call. Explain.

11.33. *Routine invocation implementation project.*
(a) Study the routine call and routine exit instructions of a machine and assembler available to you.
(b) Design an activation record layout, simple parameter passing and routine call and routine exit sequences for recursive routines, and test your design by running the translation (manual or otherwise) of, for example, the mutually recursive routines

```
void A(int i) {
    showSP("A.SP.entry");
    if (i > 0) {B(i−1);}
    showSP("A.SP.exit");
}

void B(int i) {
    showSP("B.SP.entry");
    if (i > 0) {A(i−1);}
    showSP("B.SP.exit");
}
```

Here showSP() is an ad-hoc routine which allows you to monitor the progress by showing the value of the stack pointer in some way; you will probably have to improvise. Start by calling A(10).
(c) Design a format for routines as parameters and test your design with

```
void A(int i, void C()) {
    showSP("A.SP.entry");
    if (i > 0) {C(i−1, A);}
    showSP("A.SP.exit");
}

void B(int i, void C()) {
    showSP("B.SP.entry");
    if (i > 0) {C(i−1, B);}
    showSP("B.SP.exit");
}
```

Start by calling A(10, B).
(d) Design a format for labels as parameters and test your design with

```
void A(int i, label L) {
    showSP("A.SP.entry");
    if (i > 0) {B(i−1, exit); return ;}
    exit: showSP("A.SP.exit"); goto L;
}

void B(int i, label L) {
    showSP("B.SP.entry");
    if (i > 0) {A(i−1, exit); return ;}
    exit: showSP("B.SP.exit"); goto L;
}
```

11.34. (▷795) 1. Explain how a deterministic FSA can be implemented on a compiler with tail recursion elimination, by having a routine for each state. 2. Would you recommend this technique? 3. Why doesn't this work for non-deterministic FSAs?

11.35. (▷www) Refer to the subsection on signal handlers in Section 11.4.3.2 on handling run-time errors. Consider the following code fragment in a language that allows a exception handler to resume at the point where the error occurred:

```
X := A / B;
Y := A / B;
```

An optimizing compiler wants to transform this code into:

```
X := A / B;
Y := X;
```

Explain why, if an exception handler is defined for the code fragment, this optimization is incorrect.

11.36. (▷795) A language designer is designing a programming language that features, among other things, generic routines with generic type parameters. He/she considers adding generic routines as generic parameters (non-generic routines can already be passed as normal parameters in the language) and comes to you, a compiler designer, for advice. Evaluate the proposed addition from a compiler construction point of view.

11.37. *History of imperative language implementation*: Study Sheridan's 1959 paper [260] on the IBM FORTRAN compiler, and identify and summarize the techniques used.