Eidgenössische
Technische
Hochschule
Zürich

*Berichte
des
Instituts für
Informatik*

Urs Ammann

Error Recovery in Recursive Descent Parsers

Run-time Storage Organization

Mai 1978

25

Eidgenössische
Technische
Hochschule
Zürich

*Berichte
des
Instituts für
Informatik*

Urs Ammann

Error Recovery in Recursive Descent Parsers

Run-time Storage Organization

ERROR RECOVERY IN RECURSIVE DESCENT PARSERS *

Urs Ammann
Institut für Informatik der ETH
ETH-Zentrum
CH-8092 Zürich
Switzerland

Summary:
An attempt is made to familiarize the reader with the term error recovery and to develop a simple but effective method of error recovery applicable to recursive descent parsers. Examples from a Pascal production compiler are given to illustrate the implementation of the principles worked out in the paper.

*) From an IRIA course on the "State of the Art and Future Trends in Compilation" held by the author in Montpellier (France) in January 1978; appears also in the course proceedings.

Table of Contents

Introduction

Usually the syntax of programming languages is formally defined (e.g. by means of BNF (Algol 60 [Nau63]), EBNF (Modula [Wir77]), syntax diagrams (Pascal [JeW75]) or the more powerful two-level grammars (Algol 68 [vWi69])). Programs which obey the syntactic rules are said to be syntactically correct while programs found in conflict with these rules are said to contain syntax errors. The term error recovery will be used to denote the action undertaken by the parser to assure a sensible continuation of the syntax analysis whenever a syntactic error has been detected.

Among the several methods of syntax analysis, recursive descent is certainly the most elegant one. However, in order to be applicable two prerequisites have to be fulfilled. First, the parser implementation language must allow for recursive procedures. Second, the grammar underlying the language has to be LL(1). (Following [Knu71], the LL(1) property of a grammar G = <VT,VN,S,P> can be defined as follows:

$$\forall (\alpha_1, \alpha_4, \alpha_4' \in VT^*,\ \alpha_2, \alpha_2', \alpha_3 \in (VT \cup VN)^*,\ a \in VT,\ A \in VN):$$
$$S \xrightarrow{*L} \alpha_1 A \alpha_3 \xrightarrow{L} \alpha_1 \alpha_2 \alpha_3 \xrightarrow{*L} \alpha_1 a \alpha_4$$
$$\wedge\ S \xrightarrow{*L} \alpha_1 A \alpha_3 \xrightarrow{L} \alpha_1 \alpha_2' \alpha_3 \xrightarrow{*L} \alpha_1 a \alpha_4'$$
$$\Rightarrow \alpha_2 = \alpha_2'$$

which means that when a sentential form $\alpha_1 A \alpha_3$ through the application of a production to the leftmost non-terminal A on the one hand leads to $\alpha_1\alpha_2\alpha_3$ (and finally to the sentence $\alpha_1 a \alpha_4$) and on the other hand to $\alpha_1\alpha_2'\alpha_3$ and finally to the sentence $\alpha_1 a \alpha_4'$, it follows that α_2 and α_2' are equal. In other words: the incoming symbol a leaves no doubt as to which production should be applied to the leftmost non-terminal A (it is the production $A \rightarrow \alpha_2$)).

Under these two conditions the implementation of a recursive descent parser for syntactically correct programs is trivial. Each syntactic entity (defined e.g. by a syntax diagram or a BNF non-terminal) is associated with a parser procedure. All that is needed in addition is an auxiliary procedure to scan symbols reading with each call the next terminal symbol from the input. Note that the parser obtained this way works top-down, goal-oriented, starting with the distinguished non-terminal symbol (<program>) and applying productions to the leftmost non-terminal to match the string of incoming symbols. In contrast, the symbol scanner works bottom-up, source-oriented. It reduces the character input to a sequence of terminal symbols. Figure 1 illustrates this.

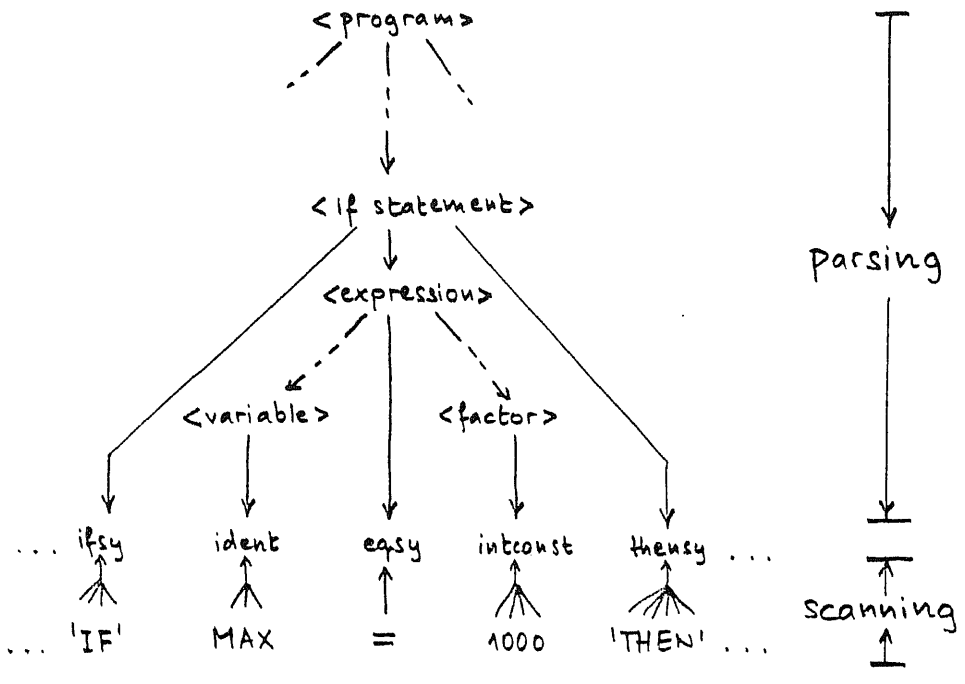


Figure 1: Parsing and scanning

With the syntax analysis separated into parsing and scanning, there are in principle two levels for attempting error correction: character level and symbol level. The character-level error repair is done in the symbol scanner. It essentially concerns the correction of spelling errors, especially in word delimiters. We do not cover this subject here. The interested reader is referred to [Gri71]. The symbol-level error recovery is done in the parser. It will be the subject of our course. (For surveys on error recovery in general see [Gri75] and [Hor75]).

A good error recovery is characterized by the fact that the parser picks up immediately after the detection of an error. This means that in general no unjustified error messages are emitted and no error escapes its detection. This allows the programmer an opportunity for a syntactically correct program after just one compilation.

When the partial syntax tree S which has already been accepted by the parser is incompatible with the incoming symbol s, an error has been found. In such a case the parser first emits an appropriate error message indicating exactly where the error was detected (cf Appendix II). Then, error recovery is initiated to solve the incompatibility. This can be done in one of the three following ways.

(1) The program is skipped up to a symbol s´ which is compatible with the actual syntax tree S.

(2) The partial syntax tree is rebuilt to form a similar tree S´ which is compatible with the incoming symbol s.

(3) A combination of (1) and (2) is used. A skip over the input up to a symbol s´ and a reconstruction resulting in tree S´ are done in tune with one another and in such a way that s´ and S´ will be compatible.

Obviously, only (3) which contains (1) (for S´=S) as well as (2) (for s´=s) is general enough to recommend itself for error recovery. In recursive descent parsers, however, it is possible that the syntax tree associated with a program is not explicitely built up during parsing: This is typically the case in one-pass compilers, in which the semantic actions therefore have to be interleaved with the parser actions. In such a compiler the syntax tree exists only implicitely, namely in the runtime stack of the compiler, which at any time represents the actual branch of the syntax tree. Figure 2 gives an example. It shows the actual branch of the syntax tree at the moment when <expression> accepts the equal sign.

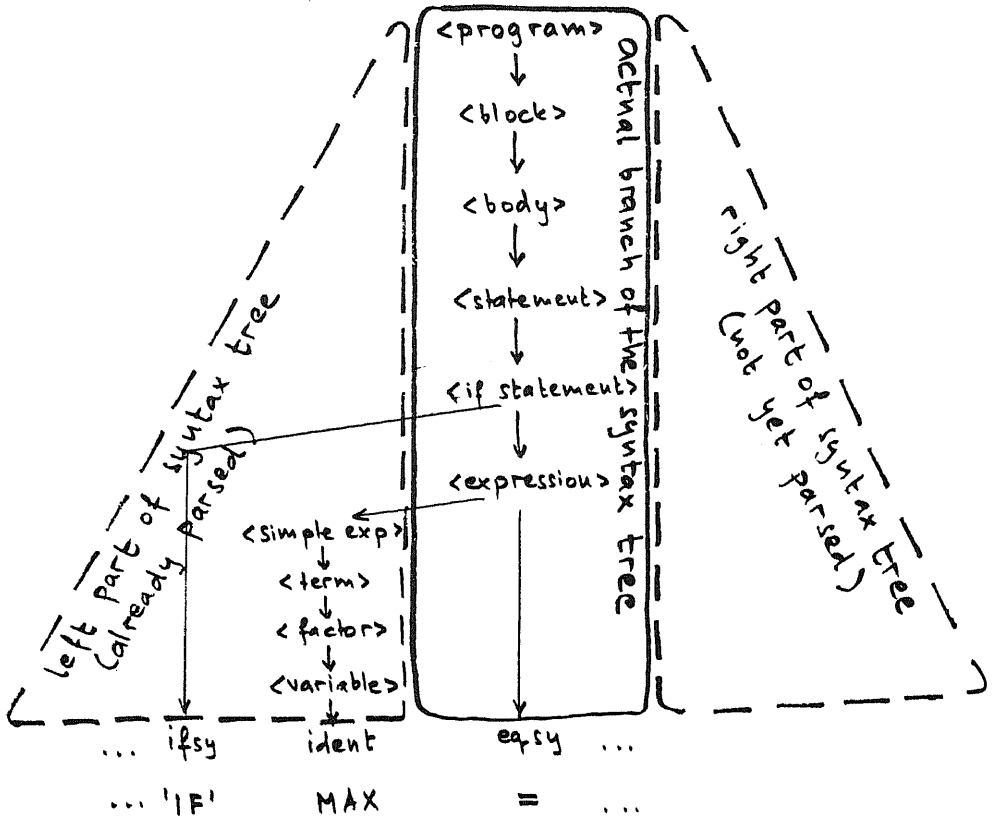


Figure 2: The actual branch of the syntax tree

Thus, in the case of a recursive descent parser the above mentioned reconstruction of the syntax tree is achieved by simply passing program control.

To illustrate this let us assume that in the example of figure 2 a for follows the equal sign. Figure 3 shows how the parser might recover from the error by passing control from <factor>, which detects the error, to <forstatement>, thereby rebuilding the (actual branch of the) syntax tree.

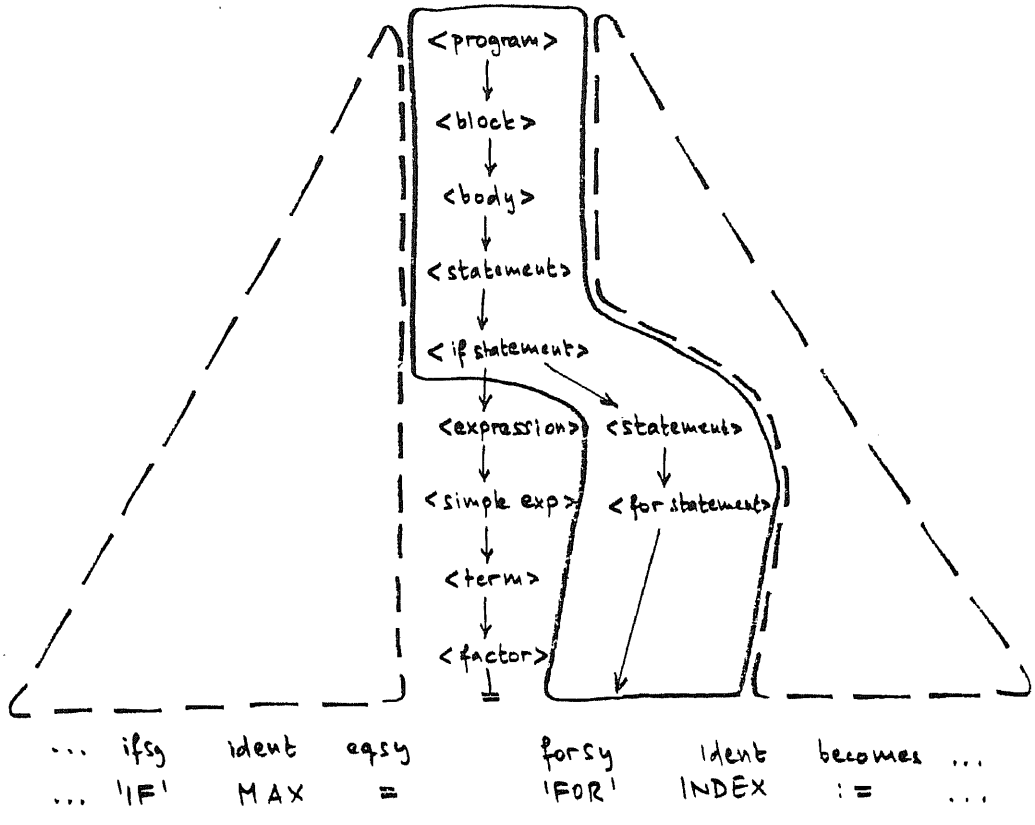


Figure 3: Reconstruction of the syntax tree to recover from a syntactic error

It is to be pointed out that passing control should never be done with a goto but rather and ultimately by using regular control structures.

One very important remark to close this introduction: Every syntactically correct program must be parsed completely, no matter how many semantic errors it contains (for example a wrong number of actual parameters in a procedure call must not result in error recovery to skip the surplus parameters). The danger of violating this principle exists whenever syntactic and semantic analysis are done in the same pass.

Development of a Method for Error Recovery

In the following the programming language Pascal [JeW75] will be used both as the language to be parsed and as the parser implementation language. The simplicity of Pascal implies its suitability in both respects. A further advantage results from the fact that the syntax is defined both by means of syntax diagrams and BNF.

We first define the symbol scanner:

```
procedure insymbol;
begin (* reads the next Pascal terminal symbol from the input;
         communication with the parser is through a global
         variable SY of
         type symbol =
                    (labelsy,constsy,varsy,...,programsy,     (a)
                     comma,semicolon,period,...,becomes,      (b)
                     ident,intconst,realconst,stringconst,    (c)
                     othersy,                                 (d)
                     eop);                                    (e)
         The type SYMBOL consists of the word delimiters (a), the
         special symbols (b), the composed symbols (c), the no-
         Pascal symbol (d) and the end-of-program symbol (e).
         OTHERSY is returned when an illegal character is found in
         the input, EOP is returned when INSYMBOL runs out of
         input.*)
end (*insymbol*)
```

The treatment of syntactic errors will be based on two trivial procedures and a set of rules every parser procedure has to obey. Let us first define the two procedures and then the rules.

The first procedure is responsible for the output of the error messages:

```
procedure error(fnr: errornr);
begin (*indicates exactly where the error associated with the error
        number FNR has been detected (cf. Appendix II)*)
end (*error*)
```

The other procedure is responsible for the error recovery and has two parameters. One of them (FSYS1) is the set of the symbols which are compatible with the current syntax tree. The other (FSYS2) is the set of those symbols s´ for which the error recovery action (3) described in the introduction is guaranteed to take place, should SY not be compatible with the current syntax tree.

The two symbol sets are of

```
   type setofsys = set of symbol
```

and the procedure reads

```
   procedure testsys(fsys1,fsys2: setofsys; fnr: errornr);
         var fsys: setofsys;
   begin if not (sy in fsys1) then
            begin error(fnr); fsys := fsys1 + fsys2;
                  while not (sy in fsys) do insymbol
            end
   end (*testsys*)
```

With the aid of this procedure, adjusting the input string after detection of an error degenerates to a call of TESTSYS.

However - and now we come to the rules - the calling parser procedure has to guarantee that the above specifications hold for the actual parameters it passes on. For FSYS2 this is evidently nontrivial since we want this set to be as large as is reasonably possible. The larger it is, the fewer symbols will be skipped, the better is the error recovery. Note that from the definition it is clear that OTHERSY will never be a member of FSYS2 (nor of FSYS1) while EOP will always be in FSYS2.

In order to keep the parser as simple as possible we further postulate that the treatment of syntactic errors should always and ultimately be done by the called parser procedure and not by the calling one as well. This has two implications for the called procedure. First, it cannot make any assumption of the input delivered to it for analysis. In particular the input can start with an arbitrary symbol. Second, the called procedure has to guarantee its caller to return a program remnant starting with a symbol from a symbol set which the called procedure receives as a parameter.

Hence, every parser procedure has a parameter (FSYS of type SETOFSYS) through which it is informed of the so called global key symbols. Besides the purpose mentioned above, this parameter also serves to tell the called parser procedure which symbols should under no circumstances be skipped during the call. Since only TESTSYS is authorized to skip, this demand is simply accomplished by always calling TESTSYS with the union of FSYS1 and FSYS2 containing all the global key symbols.

The error recovery as defined above does not only set a standard to the called procedure but also to the calling one which must always pass as parameter a set of symbols containing all its global key symbols as well as all the symbols that upon return represent a legal program continuation. However, it should be evident that due to this convention the caller has full control over the error recovery eventually taking place in the procedure it calls.

To summarize, the three recovery rules are as follows:

(1) When calling TESTSYS, respect its parameter specifications, i.e.

FSYS1: set of currently acceptable symbols
FSYS2: set of recovery symbols (for which error recovery is guaranteed to take place)

(2) Respect that error recovery is completely required from the called parser procedure, i.e.

```
procedure syntentity(fsys: setofsys);
begin (* entry: (2.1) no assumption of SY possible *)
      (* body:  (2.2) parse SYNTENTITY never skipping over
                      a symbol from FSYS*)
      (* exit:  (2.3) make sure SY ∈ FSYS *)
end (*syntentity*)
```

(3) When P(SYS1) calls Q(SYS2) then P must guarantee that any SY ∈ SYS2 will be accepted upon return from Q (it follows from (2) that SYS2 ⊇ SYS1).

It is noteworthy that the error recovery as defined in this section is very flexible since the liberty of choosing the key symbols according to the actual syntax tree opens a wide range of error treatments. In Appendix I we give two examples of parser procedures written along the guidelines summarized in this section. These examples hopefully illustrate the simplicity, flexibility and readability of our error recovery method. To show its effectiveness furthermore, a sample output from a compiler [Amm78a,b] in which this error recovery method has been implemented is also given (Appendix II).

Conclusion

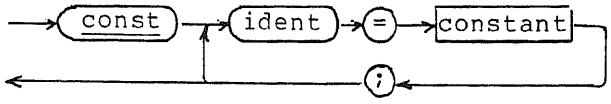
The method of error recovery described in this paper was first used in the one-pass Pascal compiler [Amm78a,b] developed at ETH in 1973 and was later adopted by many of its derivatives.

The method has proved to be successful and it could well be formalized to be used in a parser generator. However, few people ever write a compiler and nearly nobody does it more than twice. This is why for most of us the development of a parser generator is perhaps not very attractive. Furthermore, it is to be pointed out that tuning the error recovery to individual programmer errors is still best done by hand anyway.

Appendix I: Two Examples of Parser Procedures

1. The syntax of a pascal constant definition is

constantdefinition



A parser procedure respecting our recovery rules could read

```
procedure constantdefinition(fsys: setofsys);
begin (*constsy has already been accepted*)
      testsys([ident],fsys,2);                                    (1)
      while sy=ident do
        begin insymbol; testsys([equ],fsys+constbegsys,6);(2)
              if sy=equ then insymbol else error(16);
              constant(fsys+[semicolon,ident]);
              if sy=semicolon then
                begin insymbol;testsys(fsys+[ident],[],6) end
              else error(14)
        end
end (*constantdefinition*)
```

Remarks:

(1) The decoding of the error messages is as follows:

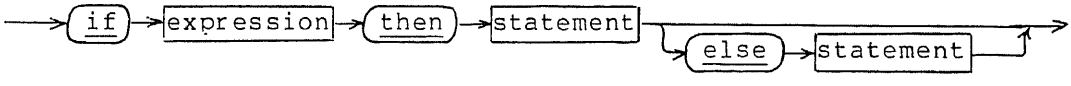
2	identifier expected
6	illegal symbol
14	';' expected
16	'=' expected

(2) CONSTBEGSYS is the set of symbols a <constant> can start with:

CONSTBEGSYS =
{x | <constant> $\Rightarrow^{*}$ xα and x$\in$SYMBOL and $\alpha\in$SYMBOL*}

2. The syntax of a Pascal if statement is

If statement



and a straightforward implementation leads to

```
procedure ifstatement(fsys: setofsys);
begin (*ifsy has already been accepted*)
      expression(fsys+[thensy,elsesy]+statbegsys);        (1)
      if sy=thensy then insymbol else error(52);          (2)
     statement(fsys+[elsesy]);
      if sy = elsesy then
         begin insymbol; statement(fsys)end
end (*ifstatement*)
```

Remarks:

(1) STATBEGSYS is the set of symbols a <statement> can start with:
STATBEGSYS =
{x| <statement> $\xrightarrow{*}$ xα and x$\in$SYMBOL and $\alpha\in$SYMBOL*}

(2) 52 'then' expected

Appendix II: A Sample Output of the Pascal 6000-3.4 Compiler

```
000006 PROGRAM EASTER(OUTPUT);
000235    CONST 1 LIM1 = 1978;  LIM2  2000;
***             ↑2                  ↑16
000235    TYPE YEAR = LIM1..LIM2; MONTH = 3..4; TYPE DAY = INTEGER;
***                                                 ↑18
000235    VAR Y: YEAR; M: MONTH; D:    ;
***                                 ↑10
000237
000237    PROCEDURE MISTERY(Y: YEAR; VAR N: DAY; VAR M: MONTH);
000006       VAR G,C,X,Z,D,E: INTEGER;
000014    BEGIN G := Y MOD 19 + 1;
000015       C := Y DIV 100 + 1;
000023       X := 3**C DIV 4 - 12;
***               ↑58
000023       Z := (8*C + 5) DIV 25 - 5;
000033       D := 5 Y DIV 4 - X - 10;
***               ↑6
000036       E := (11*G + 20 + Z - X) MOD 30;       $
***                                                 ↑6
000046       IF E < 0 THEN E := E + 30;
000047       IF (E=25) AND (G>11) OR (E=24 THEN E := E + 1;
***                                       ↑4
000057       N := 44 - E;
000061       IF N < 21 THEN N := N + 30;
000064       N := N + 7 - (D+N) MOD 7;
000074       IF N > 31 DO
***                  ↑6
000074          BEGIN N := N - 31;
***                   ↑52
000076             M := 4
000076       ELSE M =:= 3;
***               ↑13 ↑59
000100    END (*MISTERY*);
000125 BEGIN
000125    FOR Y = LIM1 TO LIM2 DO
***               ↑51
000016       BEGIN MISTERY(Y,D,M); WRITELN(Y,D,M)
000040       END
000041 END.
```

ERROR SUMMARY:
****

```
  2:  IDENTIFIER EXPECTED
  4:  ')' EXPECTED
  6:  ILLEGAL SYMBOL
 10:  ERROR IN TYPE
 13:  'END' EXPECTED
 16:  '=' EXPECTED
 18:  ERROR IN DECLARATION PART
 51:  ':=' EXPECTED
 52:  'THEN' EXPECTED
 58:  ERROR IN FACTOR
 59:  ERROR IN VARIABLE
```

References

[Amm78a] U. Ammann "The Zurich Implementation of Pascal´, in Wiley - Interscience, 1978

[Amm78b] U. Ammann "Ein paar Bemerkungen zur Implementation von Pascal", in Computer-Monographien, Hanser-Verlag, 1978

[Gri71] D. Gries "Compiler Construction for Digital Computers", Wiley, 1971

[Gri75] D. Gries "Error Recovery and Correction - An Introduction to the Literature", in Springer Lecture Notes in Computer Science, No 21

[Hor75] J. Horning "What the Compiler should tell the User", in Springer Lecture Notes in Computer Science, No 21

[JeW75] K. Jensen and N. Wirth "Pascal - User Manual and Report", Springer Study Edition 1975

[Knu71] D.E. Knuth "Top-Down Syntax Analysis", Acta Informatica, 1, 78-112, 1971

[Nau63] P. Naur et al. "Revised Report on the Algorithmic Language Algol 60", Numerische Mathematik, 4, 420-453, 1963

[vWi69] A. van Wijngaarden "Report on the Algorithmic Language Algol 68", Numerische Mathematik, 14, 1969

[Wir77] N. Wirth "Modula: a Language for Modular Multiprogramming", Software-Practice and Experience, 7, 3-35, 1977

RUN-TIME STORAGE ORGANIZATION *

Urs Ammann
Institut für Informatik der ETH
ETH-Zentrum
CH-8092 Zürich
Switzerland

Summary

The introductory part deals with abstract data types and their representation. In the main part a survey of the principles of run-time storage organization is given and possible solutions to the occurring problems are described. First, we concentrate on the run-time stack and summarize the implications to data organization resulting from the classical block structure in Algol-like procedure-oriented languages. A description of the implementation of random-order dynamic storage allocation by means of a run-time heap follows. We conclude with a few remarks on storage allocation in multiprograms. To support comprehension many examples are given.

*) From an IRIA course on the "State of the Art and Future Trends in Compilation" held by the author in Montpellier (France) in January 1978; appears also in the course proceedings.

Table of Contents

1. Introduction

One of the most important tasks a compiler system is faced with is the organization of the data in the user program. The high level of abstraction offered in languages such as PL/I, Simula, Algol 68, and Pascal implies a considerable overhead in storage management. The main problems in this respect are storage allocation, storage deallocation and access to objects which is of course highly influenced by their representation.

There are basically two approaches to storage management. The first possibility - since Algol 60 the classical one - consists in the declaration of data local to a block. The associated allocation and deallocation of memory is done automatically at block entry and block exit respectively. We shall also refer to these objects as stack objects.

The second possibility is due to the so-called dynamic objects. By means of a special-purpose language construct dynamic objects can be created under full programmer control. Deallocation of storage occupied by dynamic objects follows either one of two philosophies. According to the first philosophy the user is required to return the object to the run-time system - again through the use of the appropriate language construct. With the other philosophy it is up to the run-time system to find out when a dynamic object can no longer be accessed. It must then liberate the associated memory. We shall use the term heap object as a synonym to dynamic object.

Since block-local objects can be managed in a last-in-first-out order, storage management is not very difficult with stack objects. However, access to the objects is quite difficult especially as recursive calls allow for several instances of local data.

The opposite proves right for dynamic objects. The unpredictable order in which objects come into existence and vanish makes storage management quite difficult while access to heap objects is generally no problem. It only becomes a problem when these objects are moved in memory to realize a so-called storage compaction.

Further complications with storage administration arise when process- (task-) oriented multiprograms are to be compiled.

These are the main topics we shall elaborate in our course.

2. Abstract Data Types and their Representation

Abstract data types define a set of values an object of such a type can take on, and usually a set of operators which operate on objects of these types.

The most elementary data types are those which have unstructured values such as

type:	representation:
integer	Usually one´s or two´s complement binary representation (one or a few words in length, depending on the

underlying machine and the application).
In commercial applications an integer type is known whose values are represented by binary coded decimal digits (BCD representation, 4 bits per decimal digit).

real — Usually represented by a pair <m,e> (mantissa and exponent) of binary coded integers in one's or two's complement representation, one or a few words in length, mainly depending on the instruction set of the underlying machine. The base b of this exponential representation is usually 2, 8, or 16. Real values x are then approximated by $m*b^e$ and represented by <m,e>.

Boolean — It consists of the Boolean values false and true, for whose representation one bit is sufficient. However, for convenient access, a less tight byte (or even word) representation is used. In order to allow for efficient compilation of Boolean expressions and conditional branches, false is often represented by a zero value while a non-zero value (sometimes a one) signifies a true value.

character — Its representation is normally through binary coded integers in the range of 0..2**k-1, where k is 6,7,or 8, depending on the character set in use.

Besides these standard types, the so-called enumeration types are very important. In Pascal notation they write (<id1>, <id2>, ..., <idn>). Depending on the operations allowed one can either represent these n values by the binary coded integers 0..n-1, (which e.g. allows for an easy step through the elements), or take the internal representation of the id's (which allows for easy read-in/write-out).

The best-known structured data type is the array. It allows random access to its components through a computable index. We shall use the term static array if the number of elements is known at compile-time and dynamic array if the number of elements is only known at run-time. But, as usual, the number of elements will be limited in a way to allow the representation of the whole array in the main memory of the computer.

Mapping multidimensional arrays into the linear memory is done either row-wise or column-wise. With row-wise mapping the elements of each (hyper-)row follow one another in sequence. With column-wise mapping the same is true for every (hyper-)column. Our remarks will be based on row-wise mapping.

Let us suppose the following declaration of an array variable (Pascal notation):

```
VAR a: ARRAY [l1..u1,l2..u2,l3..u3] OF integer
```

On the assumption that the representation of an integer needs one word, the address of the element a[i,j,k] is

```
addr(a[i,j,k]) = addr(a) + ((i-l1)*(u2-l2+1)
                 + (j-l2))*(u3-l3+1) + (k-l3)
```

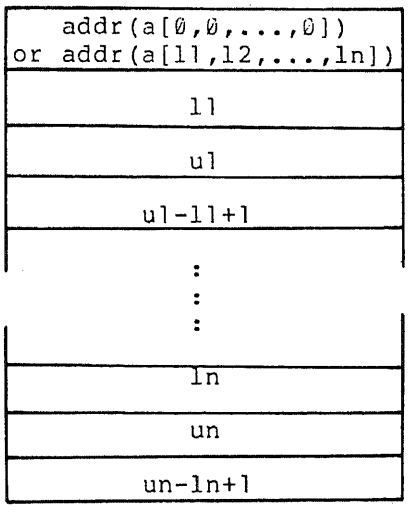
If the index limits are known at compile-time, this address calculation can be simplified in that most of it can be done at

compile-time.

```
addr(a[i,j,k]) = addr(a) - ((l1*(u2-l2+1) + l2)*(u3-l3+1) + l3)
                         + ((i*(u2-l2+1) + j)*(u3-l3+1) + k
               = addr(a[0,0,0]) + (i*c1 + j)*c2 + k
```

This shows that an element of such an n-dimensional array can be addressed with an easily calculable offset from the imaginary array origin addr(a[0,0,...,0]). The calculation of the offset needs n additions and n-1 multiplications. Note that no additional memory is required to represent a static array, i.e. an array with index bounds known at compile-time.

This contrasts favourably with the representation of dynamic arrays, i.e. arrays whose index bounds are not all known at compile-time. These arrays are usually described by what is called a dope vector. It contains the imaginary array origin (or sometimes addr(l1,l2,...,ln])) and for each dimension its low bound li, its upper bound ui and the value of ui-li+1 used in the address calculation. These values are calculated upon block entry and are then copied into the dope vector



Access to array elements is ultimately accomplished by using information from the dope vector. The ui's are part of this information area in order to allow for inrange tests on actual subscripts.

It is noteworthy that the length of a dope vector is known at compile-time (3*n+1 in the above unpacked form), since the dimensionality of a dynamic array is known from its declaration. This will later be of importance.

Another possibility to represent arrays - which can be quite effective as far as access to elements is concerned - is through what is sometimes called Iliffe vectors. Address calculation is then free of multiplications at the expense of increased storage requirements. The idea behind this method to represent arrays is to have a pointer to each (hyper-)row (or hyper-column) stored along with the array thereby providing indirect access. Figure 1 shows the representation of the array given above.

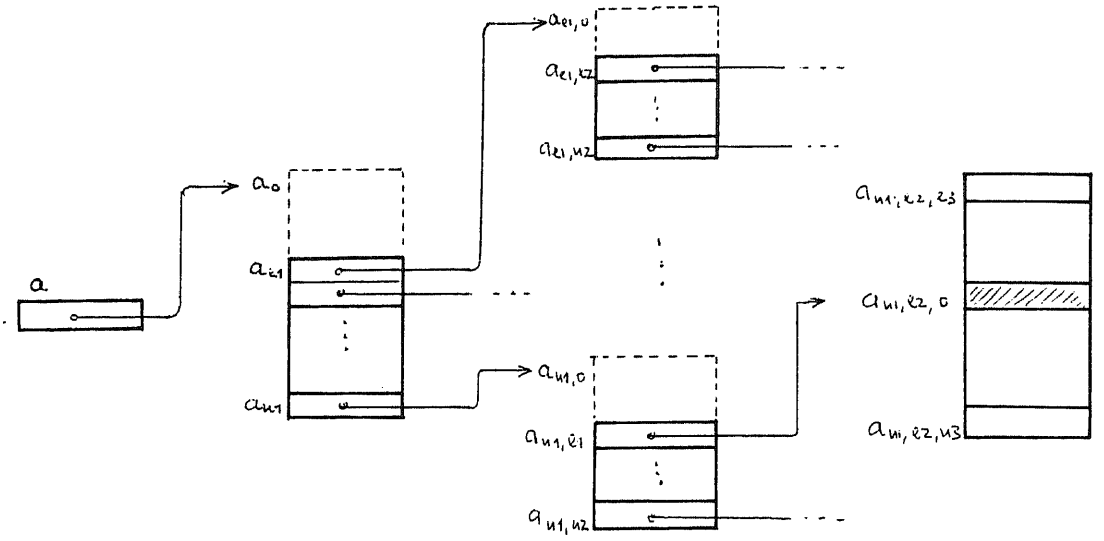


Figure 1: Iliffe vectors

Address calculation leads to

addr(a[i,j,k]) = cont(cont(cont(a)+i)+j)+k

i.e. access to an element of an n-dimensional array is n times indirect, but needs no multiplications. The additional storage requirement in our example is

1 + (u1-l1+1) + (u1-l1+1)*(u2-l2+1)

If the index bounds are not known at compile-time, this method risks to be very storage-inefficient. It also has the disadvantage of the overhead to set up the Iliffe vectors. But there are also advantages: the elements need not be stored contiguously and their access is quite efficient.

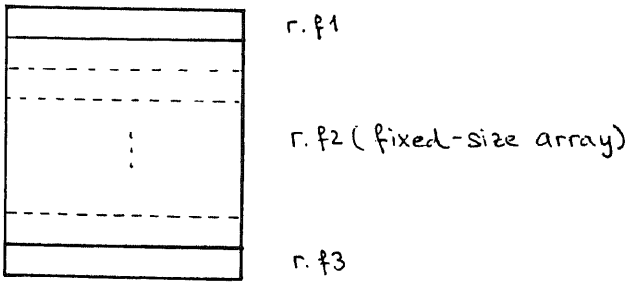
Another very well-known structured data type is what Hoare in [DDH72] calls a Carthesian product, PL/I and Algol 68 call structure, and what we shall call record (adopting the Pascal nomenclature). It allows to unite several so-called fields (of unstructured or structured type) to form an entity. Access to the fields is through the unique field names.

The representation of records most naturally uses a contiguous piece of storage.

For example, given the declaration

```
VAR r: RECORD f1: integer;
              f2: ARRAY [0..n] OF integer;
              f3: Boolean
       END
```

a possible representation is



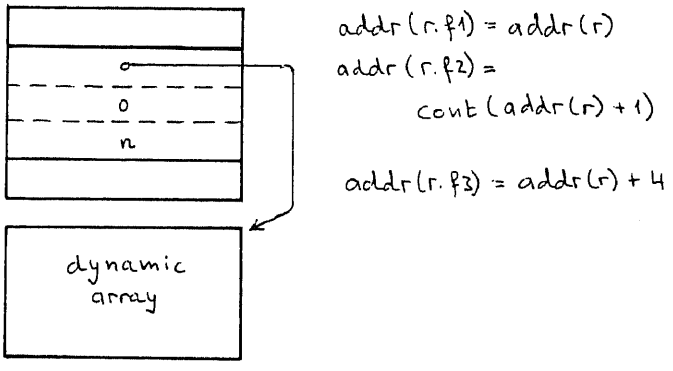
and access to the fields is relative to the record origin:

addr(r.f1) = addr(r)

addr(r.f2) = addr(r) + 1

addr(r.f3) = addr(r) + n + 2

However, if not all field sizes are known at compile-time, it is best to split the dynamic fields into a fixed part and a dynamic part, whose fixed part only (i.e. the descriptor) is stored in the record area. The dynamic part is allocated elsewhere and is referenced by the fixed part.



In this way the fields with a size known at compile-time can be accessed relative to the record origin and with an offset known to the compiler while for the dynamic fields the same proves at least correct for their descriptors.

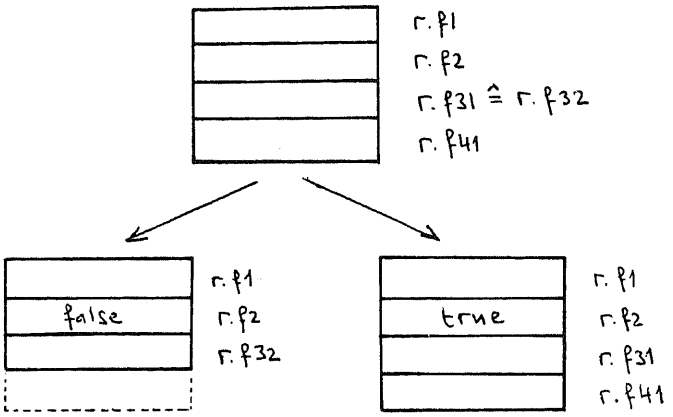
Note that records can be nested. To the compiler this poses no problems and involves no further run-time inefficiency in accessing the fields. Calculation of a field address relative to the record origin can still be done at compile-time as long as the field is not of variable size.

A generalization of the record is what Pascal calls the variant record (Hoare [DDH72] and Algol 68 use the term union instead, while PL/I calls it a cell attribute). It allows to unite several variants in one record.

Example:

```
VAR  r: RECORD f1: integer;
          CASE f2: Boolean OF
            true:  (f31: integer; f41: real);
            false: (f32: Boolean)
        END
```

The representation and access of fields is exactly the same as in the non-variant case of a record.



Hence, record variants impose no run-time inefficiency if we neglect the fact that a safe implementation has to check at run-time that any field access be consistent with the current value(s) of the tag field(s).

Other structured data types in modern programming languages such as Pascal are sets and files. We do not treat them here since this would be too far-leading.

However, a few words should be said to what Hoare calls recursive data structures (such as linear lists, sequences, queues, binary trees and others). Pascal for example allows recursive data stuctures to be defined through just one other unstructured data type, the pointer. The same proves right for PL/I and Algol 68.

By providing explicit pointer declarations Pascal makes no attempt to hide the implementation of recursive data structures (which is always through pointers) from the programmer.

Example:

```
TYPE  ptrtobinarytree = ↑ binarytree;
      binarytree = RECORD i : information;
                          l,r: ptrtobinarytree
                   END;
VAR   root: ptrtobinarytree
```

Figure 2 gives a possible representation of an instance of a binary tree.

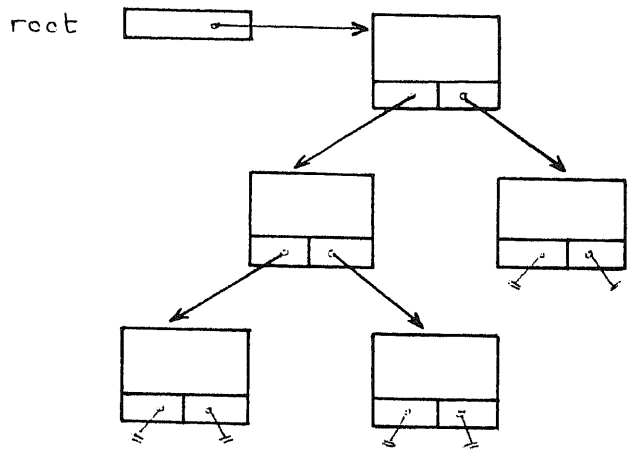


Figure 2: Binary tree representation

Since recursive objects grow and shrink dynamically during run-time, they are to be heap objects. The difficulties recursive objects involve in this respect contrast unfavourably with their unproblematic representation through pointers. We shall come back to this later.

To close this chapter, another aspect of data representation that goes across all structured data types should be mentioned. It is the question about the density of element packing.

The most efficient method with respect to access time is of course achieved by letting elements occupy easily addressable storage units (bytes, words, double words) or multiples thereof. However, this is not very efficient with respect to storage requirements.

In order to let the programmer have a certain control over what has precedence - access efficiency or storage economy - some PL´s allow to declare structured data either packed or unpacked. Pascal does so by allowing the symbol PACKED precede the declaration of a structured type. This packing option has to be seen as a compiler directive with no influence on the program itself with respect to what it computes.

Examples:

```
PACKED ARRAY [1..4,1..4] OF Boolean
```

can be packed to need 16 bits instead of 16 words, and a record such as

```
PACKED RECORD f1: Boolean;
              f2: (red, blue, green)
       END
```

could be packed to occupy not more than 3 bits.

It is noteworthy that data packing not only saves memory space, but also speeds up certain operations such as assignments between packed

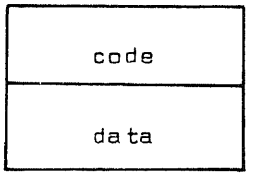
variables since fewer storage units have to be transferred.

For an excellent survey of abstract data types and their implementation the interested reader is referred to [DDH72].

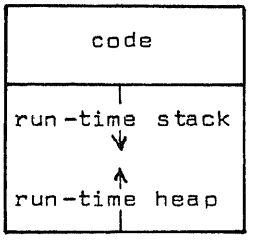
At this point we pass directly over to dynamic storage allocation since static storage allocation is untypical for modern programming languages and - an even more convincing argument - there is nothing non-trivial to be said about it!

3. Dynamic Storage Allocation

For execution, the computer operating system places a contiguous block of memory at the disposal of every user program. Usually the program is loaded into the lower part of the area, leaving the upper part to hold the data.



This data area is itself divided into two subareas. From the lower end grows the run-time stack containing the block-local data which can be administrated in a last-in-first-out order. The run-time heap grows from the upper end in the opposite direction. It holds the dynamic objects which are created and destroyed in a random order.



Obviously, a run-time test must be provided to check whether stack and heap overlap.

We shall first look at the data organization in the run-time stack.

3.1 The Run-Time Stack

It is fair to say that the classical block structure of Algol 60 has had a strong influence on modern programming languages, many of them having adopted this scheme with little or no changes. In a block-structured language we distinguish between proper blocks which are unnamed and the named parameterized blocks, the procedures. The strict static nesting of the blocks guarantees that at any time the block entered last is the one to be first left. This is the key to the last-in-first-out storage administration of block-local data, which in fact is very economical. Memory is only reserved when the associated objects come into existence (at block entry) and is

released as soon as the objects die (at block exit). In this way using the same storage locations for distinct objects becomes possible if they are not declared in nested blocks but rather in parallel blocks.

We shall first draw our attention to proper blocks and then to procedures.

3.1.1. Blocks

The block structure is a static property of programs. It is therefore known to the compiler which uses this information to overlay variables declared in parallel blocks by assigning them the same address.

Example (Algol 60):

```
BEGIN INTEGER a,b;
      :
      BEGIN INTEGER c;
            :
            BEGIN BOOLEAN d;
                  :
            END;
      END;
      :
      BEGIN INTEGER e,f;
            :
      END;
      :
END
```

Address assignment done by the compiler	
Variable:	address relative to run-time stack origin
a	0
b	1
c e	2
d f	3

Instead of 6 only 4 memory words are needed since for e/f the same storage locations can be used as for c/d.

This nearly complete resolution of blocks at compile-time is possible as long as the local data are restricted to objects with a size known at compile-time. Particularly, arrays can only be treated in this way if the bounds are fixed.

Thus, blocks with such completely static declarations involve very little run-time overhead. Only the top-of-stack register (if any) has to be updated. At block entry it has to be incremented by the number of memory words used to represent the local data, and at block exit the register has to be reset. The top-of-stack register serves both to delimit the run-time stack and to allow for push and pop operations. Figure 3 illustrates these actions.

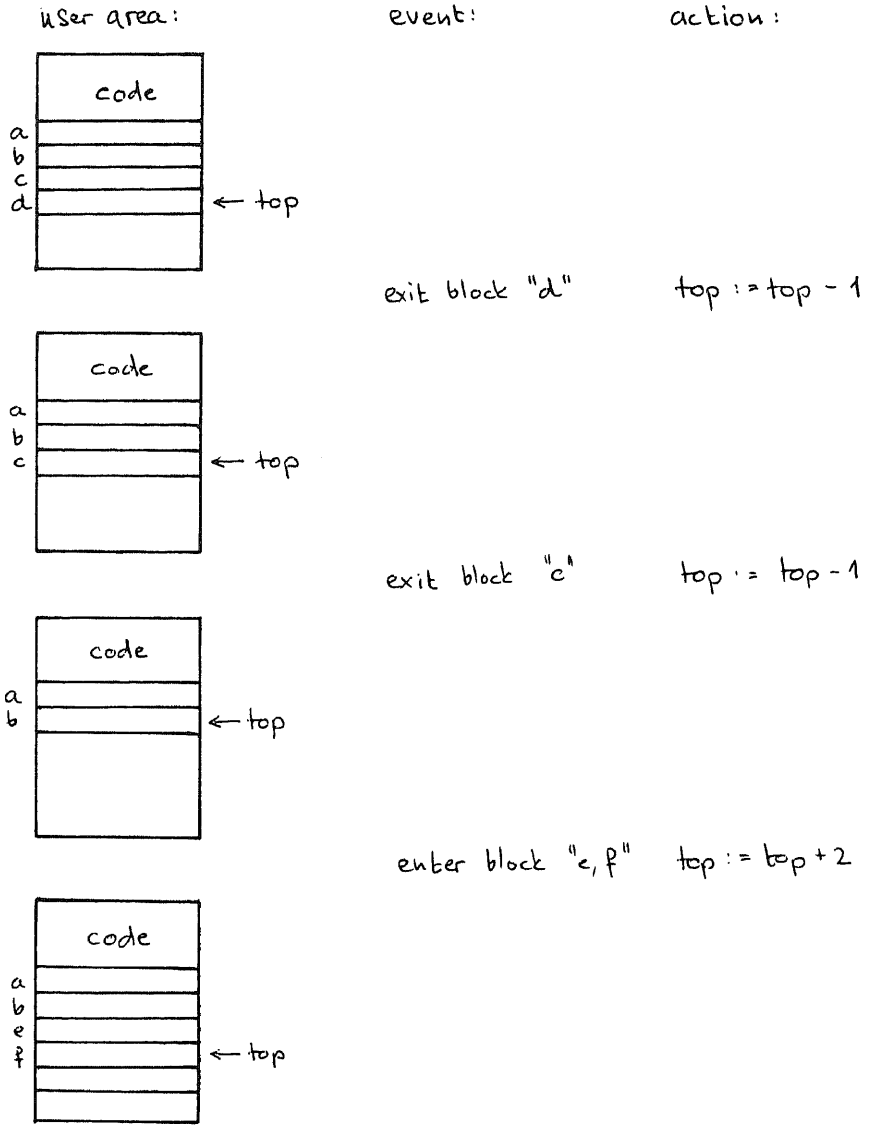


Figure 3: Stack operations

It is noteworthy that the compiler can compile implicit blocks exactly in the same way as the explicit blocks. For example an Algol 68 for loop will result in an implicit block including memory space for the loop variable, the loop bound and eventually a few auxiliary variables necessary for loop optimization.

As long as the size of each declared object is known at compile-time it is possible to assign fixed addresses (relative to the run-time stack origin) to all objects. However, one single dynamic array will imperil the quick and easy access of the data further down in the stack.

Example:

```
BEGIN INTEGER n;
      read(n);
      BEGIN INTEGER i;
            INTEGER ARRAY a[1:n];
            INTEGER j;
            :
            BEGIN INTEGER k,l;
                  :
            END;
            :
            BEGIN INTEGER m;
                  :
            END;
      END;
END
```

variable:	relative address:
n	0
i	1
a	2
j	n+2
k m	n+3
l	n+4

There are two solutions to this problem, depending on whether run-time efficiency or storage economy has higher priority. If efficiency is important, the compiler will always allocate dynamic arrays at the top of the stack to assure that at least all static variables have fixed relative addresses. For the above example this leads to

variable:	relative address:
n	0
i	1
j	2
k m	3
l	4
a	5

The penalty we pay in this example is that when we enter the second block locations 3 and 4 are reserved although they are not yet used.

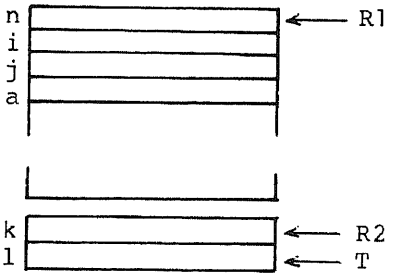
If storage economy has higher priority, one should avoid addressing all the static variables by a fixed relative address from the run-time stack origin. Instead, the compiler should provide a so-called data segment for the local variables of those blocks which are declared in blocks containing variables whose size is not known at compile-time. Local variables are then addressed relative to the segment origin.

As a conseqence, data segments generally consist of a lower part provided for the static variables and an upper part provided for the dynamic variables. This allows the static variables to be addressed with fixed relative addresses from the segment origin. The variables with varying size are addressed through their descriptors which have fixed size and therefore reside in the lower part of the data segment.

In case of our example this data organization leads to two data segments and the address assignment is as follows.

variable:	data segment number:	relative address:
n	1	0
i		1
j		2
a		3
k m	2	0
l		1

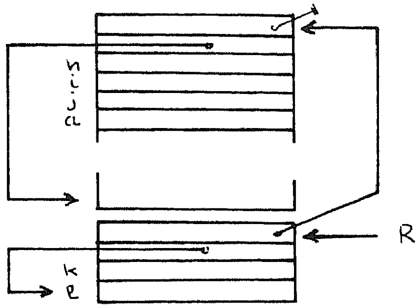
If a sufficient number of index registers is available, the compiler should in general provide variable access through these registers. This means when a block is entered to which a new data segment belongs, not only the top-of-stack register T has to be incremented, but in addition the associated index register has to be loaded with the segment origin. In our example the run-time stack shows as follows when execution is within the first of the two innermost blocks.



If registers are rare - which is usually the case - it is suitable to eliminate the T register by simply reserving a word in every data segment to store the appropriate value at block entry. This also has the advantage that at the corresponding block exit no action is necessary to reset the top-of-stack address since the data segment is erased anyway. Note that this is also true if the block is left through the execution of a goto.

Associating every data segment with an index register is still a luxury one can usually not afford. Therefore we may use only one register (R) to point to the data segment created last. In order to still be able to address non-local variables, another word is reserved in the lower part of every data segment. This word is used to hold a pointer to the origin of the predecessor segment in the stack. With the aid of this chain it is possible to access any data in the stack.

For our example, the following run-time stack results:



This might be the place to look somewhat more carefully at the problem of assigning address attributes to variables. So far the only attribute has been the relative address with respect to the data segment origin. But evidently it needs to be completed by the static level of nesting of the associated hyper-block (the term hyper is used to reflect the fact that such a block can consist of several blocks). The complete address attribute therefore is a pair <static level of nesting, relative address>, where the nesting level of the outermost hyperblock is defined to be one, and a hyper-block local to one of level n is of level n+1.

In our previous example this leads us to the following complete address attributes:

n: <1,2>   i: <1,3>   j: <1,4>   a: <1,5>   k,m: <2,2>   l: <2,3>

Access to a non-local variable v is obviously achieved by following the chain x-y times, where x is the level of access to v and y is the level of declaration of v.

To close this section, let us describe the run-time stack manipulations for this data organization more formally (M[x] denotes the memory word with address x):

(a) block entry *with* creation of a new data segment:
M[M[R+1]+1] := R;
R := M[R+1]+1;
M[R+1] := size + R + 1

(b) Block exit *with* liberation of a data segment:
R := M[R]

(c) Block entry *without* creation of new data segment:
M[R+1] := M[R+1] + size

(d) Block exit *without* liberation of a data segment:
M[R+1] := M[R+1] - size

(e) *goto* outofblock
FOR i := 1 TO segments to leave DO R := M[R];
M[R+1] := M[R+1] - $\sum_{i=1}^{\text{blocks to leave}}$ size;

Obviously, access to non-local variables has now become indirect and therefore quite slow. We shall later consider solutions to this problem.

3.1.2 Procedures

Procedures are "named" blocks that can be parameterized. Since procedures are usually called from several blocks, it is necessary to collect the local data in a data segment of its own.

When a procedure P has been called, its data segment will be on top of the run-time stack. Below it is the data segment belonging to the block B2 that called the procedure. However, this block need not be the block B1 in which the procedure was declared (cf. figure 4)!

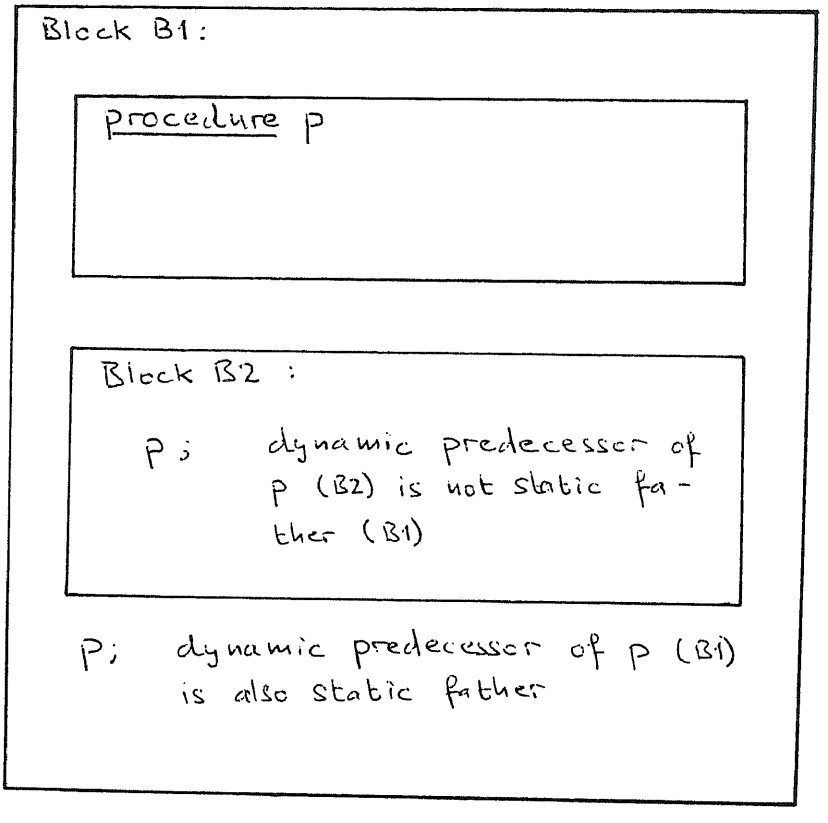


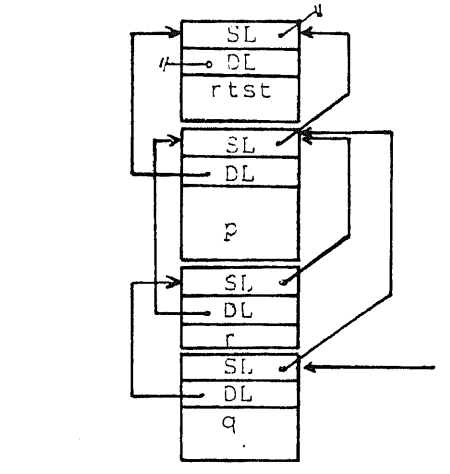
Figure 4: Dynamic predecessor and static father

Since P must have access to the variables of B1, it is up to the compiler to provide for it. This means that in the run-time stack we must now distinguish between a dynamic chain and a static chain. The dynamic chain links each data segment to its predecessor in the stack. It reflects the dynamic sequence of block entrance and is used to push and pop data segments. The static chain links the data segment of a procedure P with the data segment of the block in which P was declared. Thus, the static chain mirrors the static block structure and is used to access non-local variables.

Instead of only one we now need two pointers in the head of our data segments, the static link SL and the dynamic link DL.

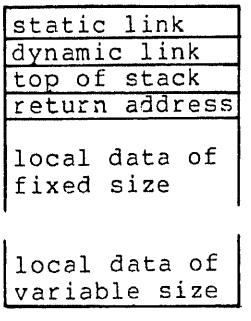
Example (Pascal):

```
program rtst(output);
   procedure p;
      procedure q;
      begin
      end;
      procedure r;
      begin
           q
      end;
   begin
        r
   end;
begin
     p
end.
```



stack of data segments belonging to the calling sequence rtst -> p -> r -> q

Considering that the return address also has to be stored in the run-time stack, a procedure data segment takes its final form:



Let us again describe the stack manipulations somewhat more formally.

```
(a)  Procedure call:    R1 := "Static Link";
                        R2 := "Return Address";
                        "jump to procedure entry"

(b)  Procedure entry:   M [M[R+2]+2] := R;
                        R := M[R+2]+1;
                        M[R] := R1; M[R+3] := R2;
                        M[R+2] := R+3+size

(c)  Procedure exit:    R1 := M[R+3];   R := M[R+1];
                        "jump to R1"
```

If non-local variables are ultimately addressed by following the static chain, a considerable inefficiency will result. It is therefore wise to hold at least the origin of the run-time stack in another register to allow direct access to the global variables. Furthermore, when consecutive accesses to variables of intermediate level occur, the static link should be followed only once. This can be achieved by immediately copying the origin of the intermediate segment into a register. Subsequent accessses will then be direct as well. This technique reduces indirect accesses considerably. A compiler in which this addressing strategy is used is described in [Amm77].

Another possibility for addressing variables - the display method - avoids multiple indirectness completely. In this method the origin of the currently accessible data segments are copied into a contiguous piece of memory, the so-called display, where they are directly accessible. This limits the access of variables to one indirectness (using the appropriate display entry). The penalty one has to pay for this gain in efficiency is the update of the display at procedure entry and exit, which in case of a formal call is non-trivial.

To avoid updating the display, other implementations give each procedure its own display. It is put into the local data segment and is initialized at procedure entry. However, this overhead may well not pay back, especially if only few non-local variables are subsequently accessed.

Procedure Parameters

They are the means to parameterize named blocks. The implementation of procedure parameters is done by passing for each actual parameter a so-called parameter descriptor. These descriptors are either passed in registers or on stack.

Sometimes - especially in FORTRAN implementations - a descriptor area is set up. This can be done at compile-time, if descriptors are addresses and these addresses are constant as they usually are with purely static storage allocation. Upon call, the first word address of the descriptor area is then passed to the called procedure.

The more compact the descriptors, the more efficient is the call. Parameter descriptors are therefore normally one word quantities.

The following parameter mechanisms can be distinguished:

(a) Call by reference: The actual parameter must be a variable whose

address is evaluated once upon the call and passed as a descriptor to access the actual parameter indirectly.

(b) Call by result: The actual parameter must be a variable whose address is evaluated once upon the call and passed as descriptor. The formal parameter is treated like a local variable, the only difference being that upon exit its value is assigned to the actual parameter (through the descriptor).

(c) Call by value: Such a parameter is nothing else than an initialized local variable. The actual parameter merely serves to determine the initial value of the local variable. There is absolutely no way for the procedure to change the value of the actual parameter.

If the size of the parameter is only one word, the descriptor can well be the value itself. Otherwise it is better to let the descriptor be the address of the actual parameter. For example if the actual parameter is a static array, we would pass its first-word-address, if it is a dynamic array, we pass the address of the array descriptor (dope vector) and in case of a string constant we would pass the address of where it is stored.

Passing only an address allows us to generate the necessary and sometimes cumbersome copy code just once - within the procedure - instead of with each call.

A variation of this kind of parameter is the so-called constant parameter which is more restrictive in that it forbids assignments to the formal parameter.

(d) Call by name: This is by far the most inefficient parameter mechanism. Every occurence of the formal parameter is textually substituted by the corresponding actual parameter expression. For the implementation this means that the actual parameter has to be evaluated each time the corresponding formal parameter occurs. Implementation is through something very much like a function procedure, the so-called thunk, which is called on every time the name parameter is referenced.

The difference between the messy name parameter and the reference parameter is nicely illustrated by the well-known example of swapping the values of two variables:

```
PROCEDURE swap (VAR  x,y: real);
   VAR z: real;
BEGIN  z := x; x := y; y := z  END;

{assume  i=3 AND  a[3] = 5}
swap (i,a[i])
{is now i=5 AND a[3] = 3?}
```

With reference parameters x,y the procedure body interpretes

```
BEGIN  z := i; i := a[3]; a[3] := z  END
```

which in fact leads to i=5 AND a[3]=3, while with name parameters the interpretation is

```
BEGIN  z := i; i := a[i]; a[i] := z  END
```

which leads to i=5 AND a[5]=3!

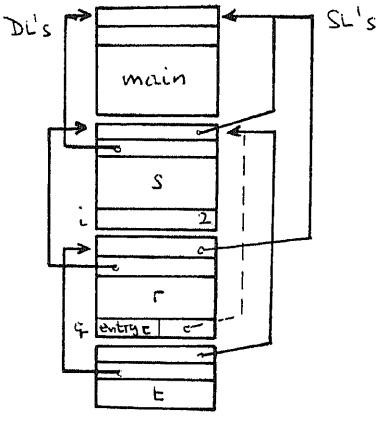
In most programming languages the parameter kind is bound to its formal parameter and cannot be altered during execution. However, in

PL/I this is not the case. In this programming language the programmer can choose upon call which kind of parameter should be applicable during the call. For example p(i) implies a reference kind parameter while p((i)) implies a value kind parameter. Needless to say that such a flexibility leads to considerable run-time inefficiency and insecurity!

The above enumeration of parameter kinds is not complete. A parameter can also be a procedure. Its descriptor consists of entry address and static link and allows to compile a call to the formal procedure very much the same way as a call to an actual one.

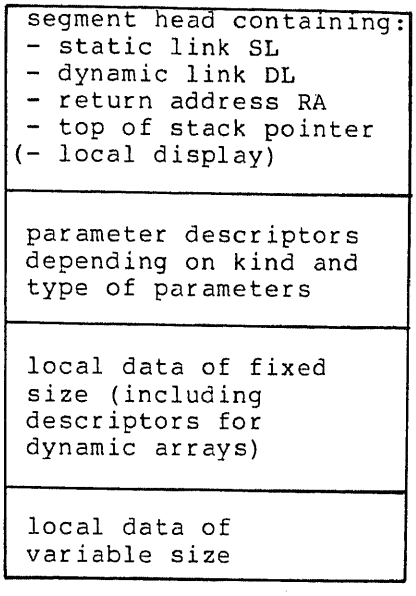
The following example shows how a parameter procedure (formally p, actually t) accesses - within another procedure (r) - a variable (i) declared in a parallel block. It also gives a picture of the state of the run-time stack during execution of the parameter procedure.

```
PROGRAM main(output);
    PROCEDURE r(PROCEDURE q);
        q
    PROCEDURE s;
        VAR i: integer;
        PROCEDURE t;
            i := i+1
        i := 1; r(t)
    s
```



Some programming languages allow formal labels. They are usually implemented through a thunk which passes control to the actual label. The side-effect of the execution of the thunk is that the run-time stack is updated appropriately.

To close this chapter on the organization of the run-time stack, the most general form of procedure data segment is given below:



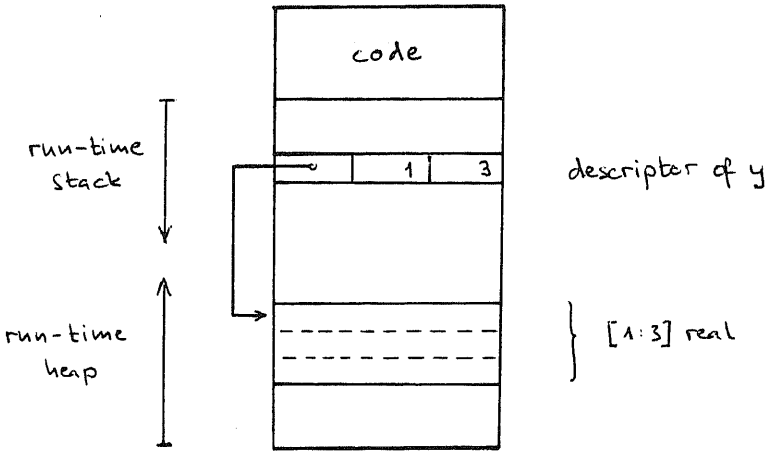
3.2 The Run-Time Heap

As mentioned in the introduction the purpose of the run-time heap is to hold the objects created during execution under programmer control. In Algol 68 such objects are identified by the generator HEAP, in PL/I they have the storage allocation attribute CONTROLLED and are called into existence by the command ALLOCATE. In Pascal the standard procedure NEW serves the same purpose.

To take an example from Algol 68 let us assume the declaration

HEAP [1:3] REAL y.

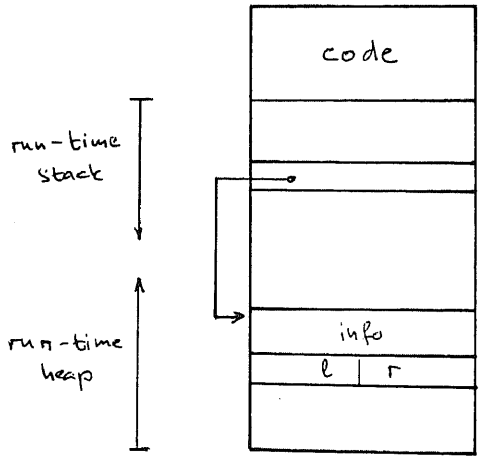
When the block is entered to which this declaration is local, an area will be reserved in the run-time heap to hold the three components of the real array. At the same time its descriptor will be allocated in the run-time stack and will be initialized:



A second example is taken from Pascal. We assume the declaration

```
TYPE ptr = ↑btree;
   btree =  RECORD i: info;
                   l,r: ptr
            END;
VAR root: ptr
```

When the block is entered to which this declaration is local, memory will be allocated to hold the value of ROOT. A later call NEW(ROOT) will then reserve a record of type BTREE in the run-time heap and will initialize ROOT to reference this record:



Obviously, access to the heap objects is always indirect.

When a block is left the local stack objects vanish. In our examples this would be the case for the descriptor of Y and ROOT respectively. However, this does not necessarily imply the disappearance of the heap objects, since meanwhile copies of their references might have been made.

As to the deallocation of heap objects, there are basically two

philosophies:

philosophy 1: The deallocation of heap objects is explicit through special purpose language constructs (e.g. FREE in PL/I, DISPOSE in Pascal).

philosophy 2: The deallocation of heap objects is done implicitly by the supporting run-time system as soon as it realizes that an object cannot be accessed any more (e.g. in Algol 68).

We shall first draw our attention to philosophy 1 and later to philosophy 2.

After a program has been in execution for a while, its associated storage area would probably look as illustrated in figure 5.

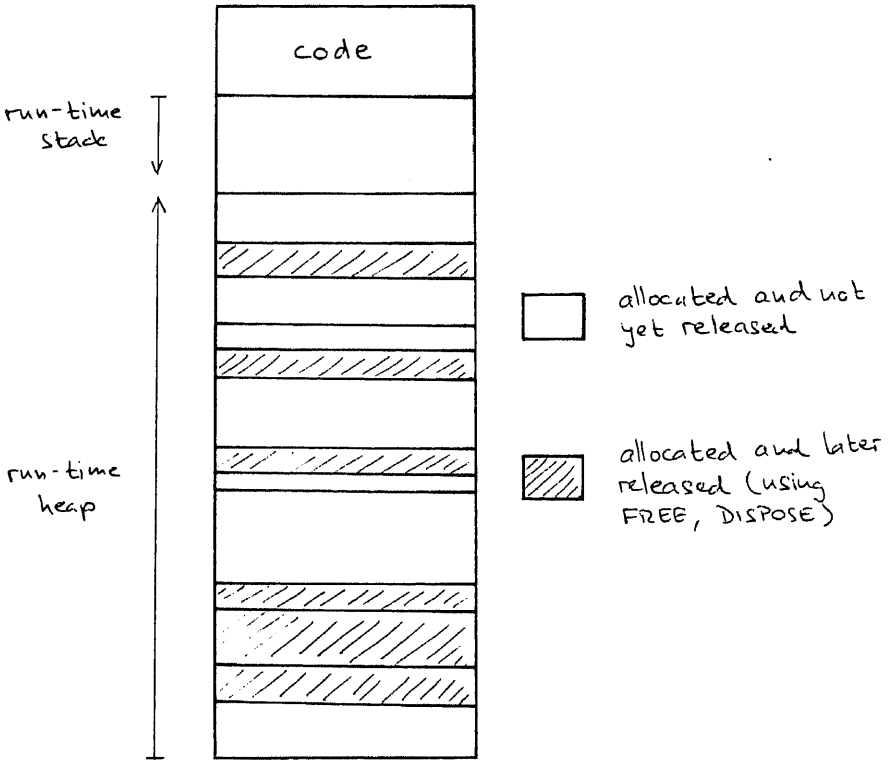


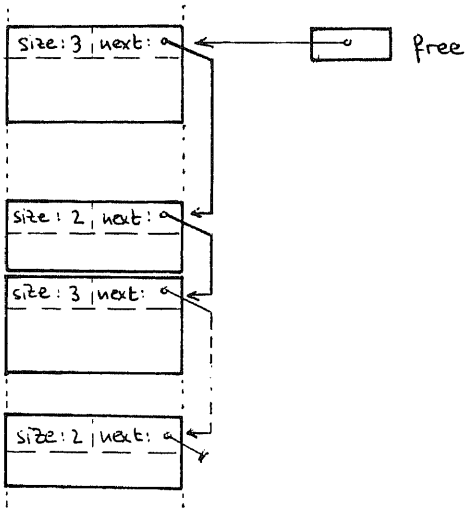
Figure 5: The run-time heap

It shows a run-time heap full of holes that risks to run into the run-time stack. Two questions now arise

i) How do we keep track of the holes within the heap?

ii) What is a good algorithm to find a block of n consecutive free words in the run-time heap and to reserve it (to be used with ALLOCATE and NEW)?

The answer to the first question is quite simple. We keep a list of available space, most conveniently and economically by using the available space itself:



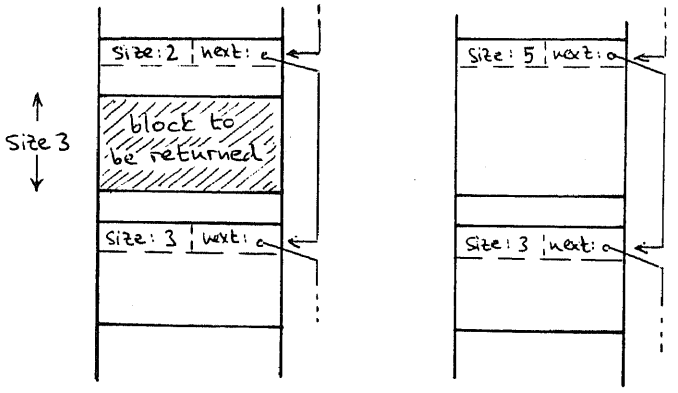
This list can be in random order or it can either be sorted with respect to the size of the free blocks or with respect to their memory addresses.

As to the second question, one is tempted to go through all the free blocks and choose the one for which size-n is a non-negative minimum. However, this so-called best-fit method is considerably less efficient than the first-fit one. With this latter strategy the search is ceased as soon as a block with size≥n is found. Its size is decreased and if it is now zero, the block is removed from the list of available space.

The serious advantage of the first-fit over the best-fit - besides its efficiency - is that there is less tendency to favouring the creation of small blocks (which can later hardly be used).

However, the first-fit clearly tends to cluster the small blocks right in front of the list. A significant improvement is therefore achieved by making the list of available space circular and starting the search always where it had been terminated the previous time.

When blocks are returned to the system a collapsing problem can arise. If the block to be returned is adjacent to one already in the free list the two blocks must be merged into one:



A possible liberation algorithm is illustrated in figure 6. It searches through about half of the free list, which must be in the

order of increasing addresses.

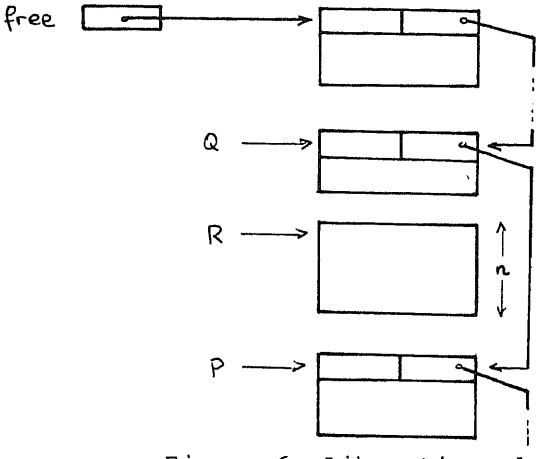


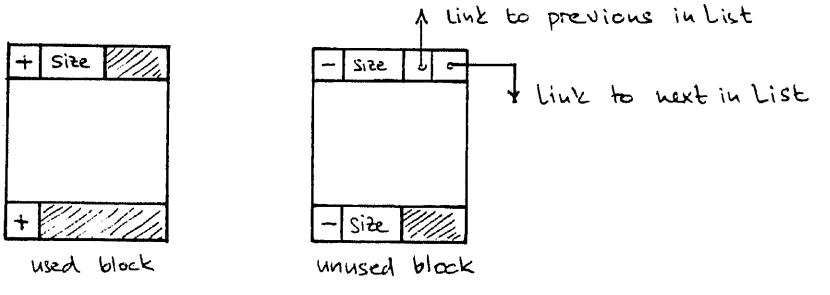
Figure 6: Liberation algorithm

Starting at FREE we go down the list of available space with two pointers P and Q, where P immediately preceeds Q. When P has become greater than the address R of the block to be returned, the place of insertion (between Q and P) is found. Now, the upper bound collapsing condition R + n = P is checked and eventually the two blocks are merged. Then we check for a lower bound collapse (Q + size(Q) = R) and eventually merge again.

Searching through approximately half the list makes this algorithm rather inefficient and leads to the question whether there is a way to maintain a list that allows liberation without searching.

In fact there is such a solution, the so-called boundary tag method. The penalty in space one has to pay for is two "words", one in front of each block and one at the end. This storage inefficiency is sometimes immaterial, but all too often unacceptable.

Used blocks are marked with a "+" and have their size in the front word while unused blocks are marked with a "-" and have their size in both head and tail word. Unused blocks form a doubly linked list.



Given this organization of the run-time heap, the boundary tag algorithm is straightforward. For details, the interested reader is referred to Knuth's excellent survey on Dynamic Storage Allocation in [Knu71].
There remains the question as to what should be done when the user asks for an area whose size exceeds the size of any element in the list of available space. Since the main point behind philosophy 1 is that the run-time system does not have to know the references to the

heap elements, it would not be consistent with this philosophy to expect that elements be moved in the heap. Instead, the above case should preferably result in a run-time error indicating heap overflow.

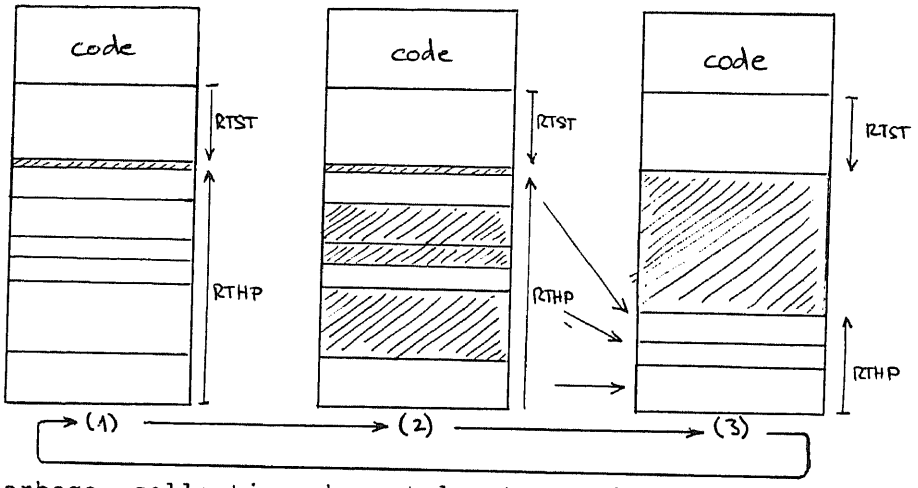
With philosophy 2, however, the situation is quite different. In order to be able to do the garbage collection, the run-time system needs to know where the references to the heap objects are located. It can therefore also move the alive objects with modest additional overhead. Hence, a possible three-stage cycle for storage allocation in the run-time heap based on philosophy 2 would be:

(1) Use run-time heap as second stack until no memory is left between stack and heap.

(2) Do garbage collection.

(3) Compact alive objects at the bottom of the heap.

An illustration of this allocation cycle follows.



Every garbage collection has at least two phases, a marking phase and a collection phase. During the marking phase accessible objects are all marked. This is done by following the alive references to the heap objects. In the collection phase the garbage (i.e. unmarked objects) is collected and the mark bits of the alive objects are reset.

The problems with garbage collection can be summarized as follows:

- It is non-trivial, especially if the heap objects are not of a unified type and can themselves reference heap objects. Under such circumstances GC may be very time consuming. In addition, its success can never be guaranteed.

- GC uses memory (for code as well as for data) exactly at a moment at which memory is extremely rare.

- At run-time, a complete list of the locations containing the references to the heap objects is needed. In general this list changes dynamically.

- Mark bits are needed, either along with each object or,

preferably, one bit per storage entity in a special mark table.

- At any time, the references to the heap objects must be well defined. Otherwise, the effect of the GC algorithm is not predictable.

This list indicates that there are a couple of quite annoying problems in connection with GC the solution to which varies from application to application.

In order to give an impression of how a GC algorithm could work in a Pascal environment, we shall apply a four phase GC algorithm to a trivial sample program. The four GC phases are the following:

(1) Marking phase: By following the dynamic chain all data segments in the stack are visited. In the head of each segment a reference to a list (which was constructed at compile time) is found (cf figure 7). This list tells where in the segment the pointers are located and to what type of objects they point. These pointers are traced one after the other under careful prevention of getting into a loop (due to a cycle). For every heap element reached in this way, the corresponding entries in the mark bit map are set.

(2) Map phase: Using this bit map a table is constructed which maps old addresses of objects onto new ones.

(3) Update phase: The paths of (1) are followed again to update each reference according to the address mapping table (2).

(4) Compacting phase: The objects are moved as defined by the address mapping table.

The data structures and variables underlying our example are

```
TYPE inforec = RECORD i: info;(*size = 2 storage units*)
                      p: ↑inforec;
               END;
     listel = RECORD ir: inforec;
                     nxt: ↑listel;
               END;
VAR fst: ↑listel; k: integer; lst: ↑listel;
```

Furthermore, we assume that, after our program has been in execution for a while, its associated storage area looks as depicted by figure 7.

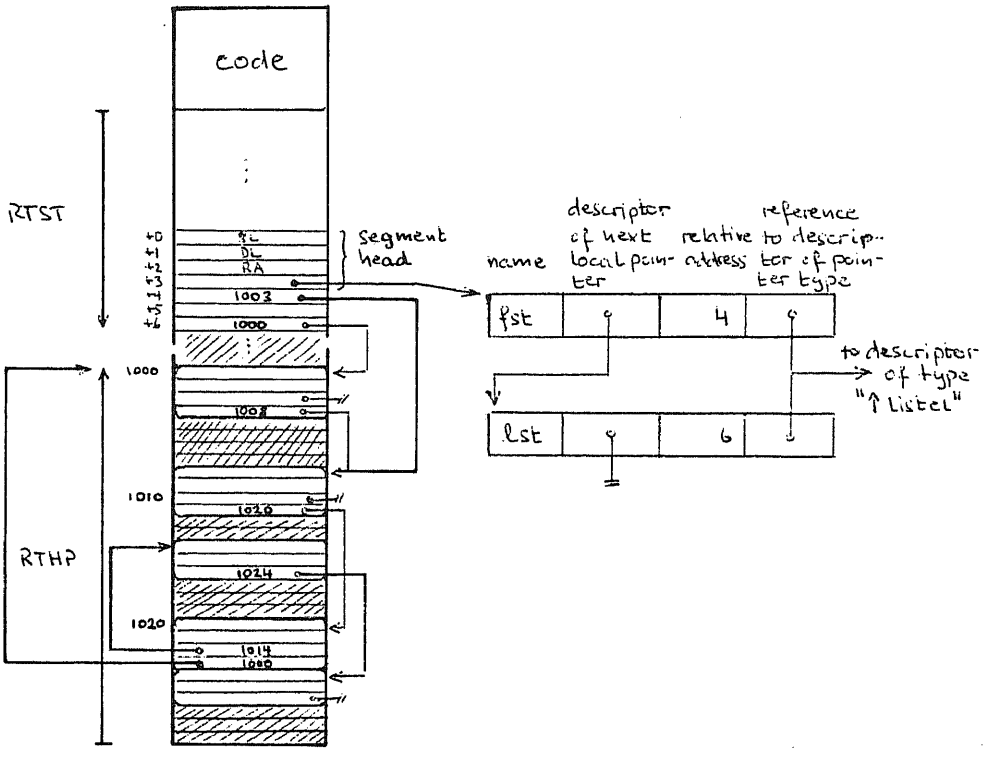


Figure 7: GC example

It shows a circular list of actually three elements hanging from its root FST. The element type is LISTEL. One of these three elements has a two element extension, each of type INFOREC. The linear list of the descriptions of the local pointer variables can be reached from the pointer stored in the segment head (relative address 3). For each pointer, its name, its offset relative to the segment origin and a pointer to the description of its type are given. Note that this information - which is assembled completely at compile-time - is not only vital for GC, but also for any sensible post mortem dump facility. The existence of a dynamic chain allows the garbage collector to visit all the data segments in the run-time stack. In each segment the entry with relative address 3 points to the description of the local pointers. It allows the garbage collector a correct interpretation of the user data area, hence also the generation of a bit map distinguishing between used and unused storage entities. In the case of our example, the following bit map would result from GC phase (1).

bit map:

memory address	1000	1001	... ...	1029
bit map element	1	1	11000011111001110001111111100	0

Ones characterize used storage entities while zeros indicate unused

locations.

In GC phase (2), this map is now processed backwards to generate a table mapping old heap addresses onto new ones. This leads to the following table:

old address(oa)	new adress	number of words to be moved
$\geq$ 1020	oa+3	7
$\geq$ 1014	oa+6	3
$\geq$ 1008	oa+8	4
$\geq$ 1000	oa+12	4

With the aid of this information the old pointer values are now updated in GC phase (3). This leads to the storage area depicted in figure 8 (left).

Finally, the alive heap elements are copied towards the higher end of the heap in GC phase (4) (figure 8, right). Again bit map and mapping table provide for the necessary information.

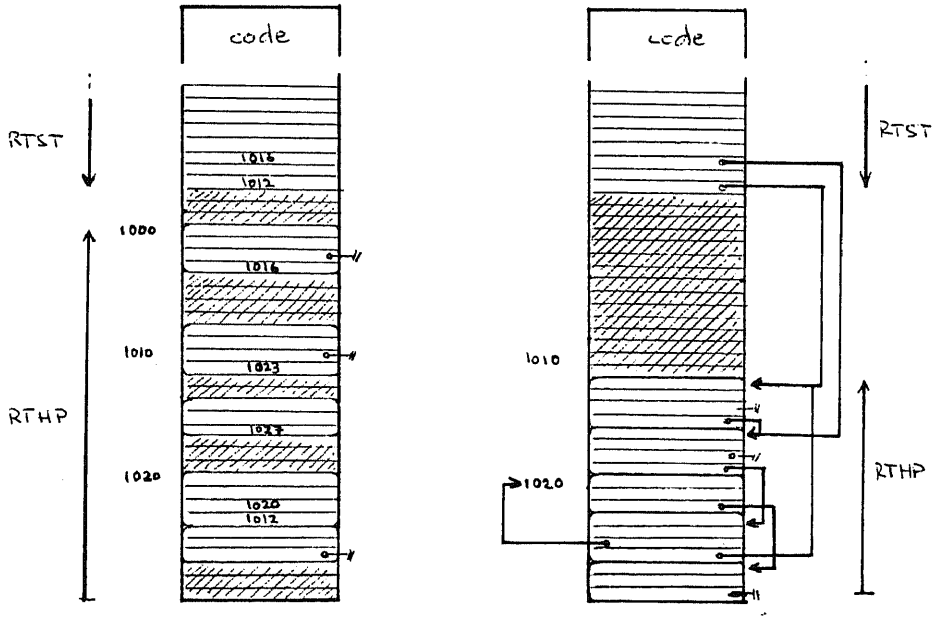


Figure 8: Example of GC phases (3) and (4)

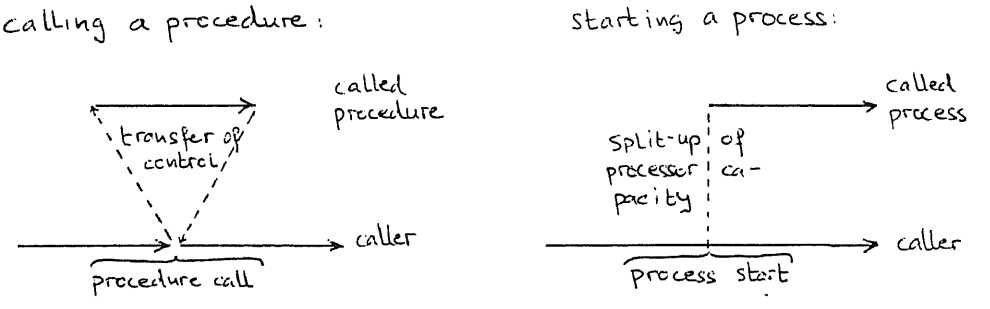
Execution of the interrupted program can now continue with a heap which is completely free of garbage.

Obviously, such a garbage collection can be very time and memory consuming. The above example of a four phase GC should therefore not

be misunderstood as a "real life" algorithm. It was reported here just to give an impression of what GC and its problems are. In fact, garbage collectors are always tailored to their specific applications. This results in very different and sophisticated algorithms, whose description would burst the scope of this survey.

3.3 Multiprograms

Several modern programming languages allow for the possibility to specify parallel computations (e.g. Algol68 through the par clause, PL/I through the CALL attribute). Adopting the philosophy of Modula [Wir77] we say that in such a multiprogram several processes run in (pseudo-)concurrency. Processes are declared in the same way as procedures. However, the effect of their activation is quite different from the activation of a procedure. In the former case we speak of starting a process, while the latter case is known as calling a procedure. Calling a procedure implies a temporary transfer of control only, while activating a process calls for an additional processor to execute the started process - or at least asks for splitting up the available processor capacity to serve the new process pseudo-concurrently with the other processes:

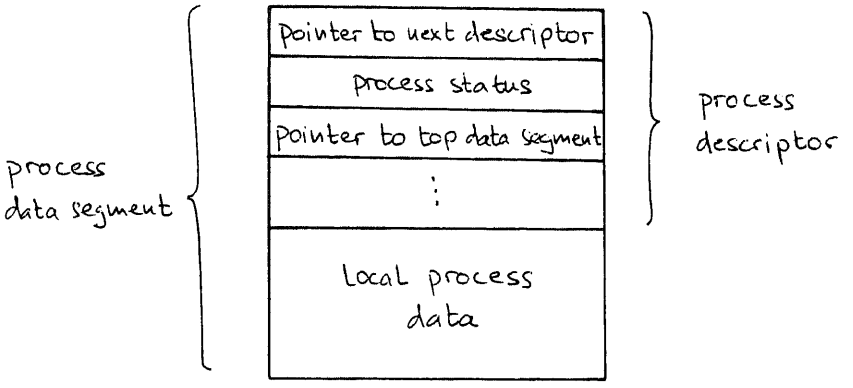


When a procedure comes to its end, control simply returns to its caller. In contrast, a process reaching its end goes out of existence (it dies) and leaves the associated processor at the disposal of the remaining processes.

Every process in a multiprogram can be viewed as a monoprogram of its own, requiring a run-time stack and a run-time heap. The "sum" of all these process-associated run-time stacks results in a forked stack. Starting a process forks the stack. Obviously, the image of this cactus stack in the linear computer memory is far from being last-in-first-out. When a process dies, it is in general not even possible to release the associated data segment, since a son process (i.e. a local process started either directly or indirectly by it) might still refer to the data of the dead process. For an example see figure 9: If the first instance of p dies before q, the data segment of p must not be discarded from the (forked) stack, because this segment is not only a member of the stack belonging to p, but also a member of the stack of its son process q. Clearly, data segments must be discarded from the forked stack only if there is no alive reference to the segment. With this respect the process data segments of a multiprogram resemble the heap elements of the previous chapter very much. It is therefore quite natural to use the storage allocation and liberation principles described earlier not only for heap objects, but also for the data segments of a

multiprogram.

To this end processes could be described by the following process descriptors (for the description of a concrete implementation cf. [Wir77]):



The descriptors are linked to form a circular list, the process ring. One descriptor entry therefore is a pointer to the next descriptor. Another entry defines the status of the process and distinguishes for example between running, ready (to run as soon as a processor becomes available), waiting (for an event to occur), and terminated (i.e. dead). A third entry points to the top segment of the process in question and hence is the head link of its dynamic and static chains.

Figure 9 gives an example of a user program with four concurrent processes and depicts the state of the associated data area after the second instance of p has been created and started. The first instance of p is currently executed by the single processor.

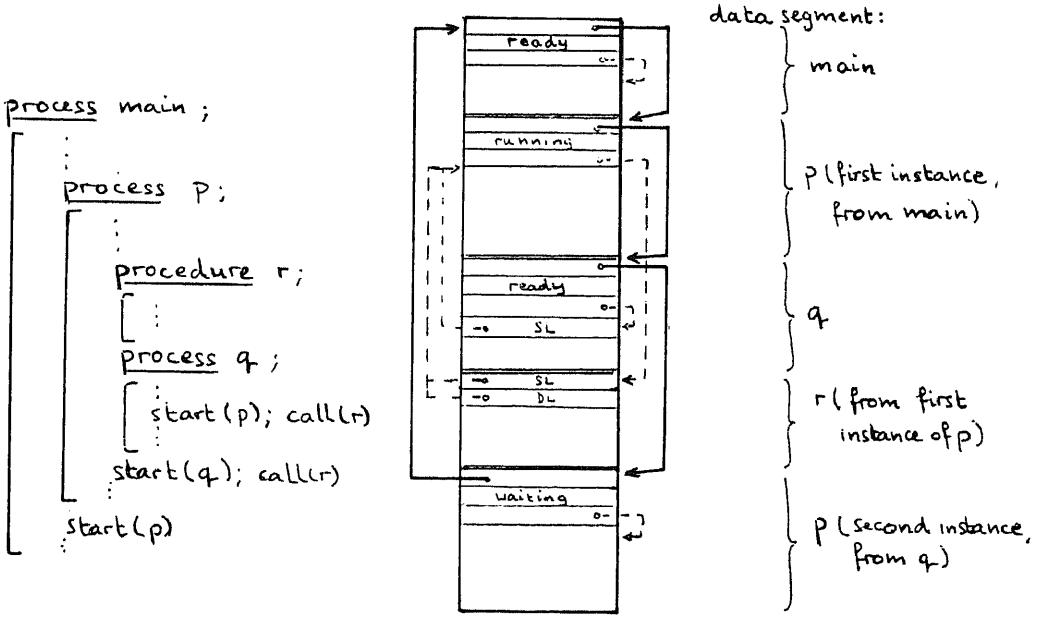


Figure 9: Example for the data organization in a multiprogram

Let us assume that this process now dies (after returning from its call to r). As mentioned before, its data segment must not be returned to the system since its son process q still refers to it.

In general, the deallocation of process segments will follow either one of the two philosophies described in the previous chapter. If philosophy 1 is chosen, a system support routine will be called each time a process dies to decide whether sons still refer to it. If none refers to it, an implicit call to the free list algorithm will return the segment to the free list. Previously dead father processes which thereby lose their last reference will be treated in the same way. With philosophy 2 the GC algorithm will have to decide which process data segments are no longer referenced. As a prerequisite the GC algorithm must have access to all the process descriptors, which is guaranteed since they form a circular list.

Conclusion

Data and storage organization in user programs is one of the most important tasks a compiler writer is faced with. It is his responsibility to fill the considerable gap between the high level of abstraction present in modern programming languages and the all too often primitive hardware on which the compiled program has to run.

However, the problem is not just to find a possible implementation for abstractions such as data structures, block-local data, dynamic data allocation, recursion, and multiprogramming. The problem is to find efficient implementations, efficient both with respect to run-time and storage requirements. Succeeding in finding whem mean not less than blowing away the last argument in favour of assembly language programming!

References

[Amm77] U. Ammann, "On Code Generation in a Pascal Compiler", Software - Practice and Experience, 7, 3

[Amm78] U. Ammann, "The Zurich Implementation", in "Pascal - The Language and its Implementation", Wiley Interscience (to appear)

[DDH72] O. -J. Dahl, E. W. Dijkstra, C. A. R. Hoare "Structured Programming", Academic Press

[GoH74] G. Goos, J. Hartmanis, (eds.) "Compiler Construction", Springer Lecture Notes in Computer Science, No. 21

[Gri71] D. Gries, "Compiler Construction for Digital Computers", Wiley

[Knu71] D. E. Knuth, "The Art of Computer Programming", Vol.1, Addison-Wesley

[Wir77] N. Wirth, "Modula: a Language for Modular Multiprogramming", Software - Practice and Experience, Vol.7, 3-84

Berichte des Instituts für Informatik

*	Nr. 1	Niklaus Wirth:	The Programming Language Pascal
*	Nr. 2	Niklaus Wirth:	Program development by step-wise refinement
	Nr. 3	Peter Läuchli:	Reduktion elektrischer Netzwerke und Gauss'sche Elimination
	Nr. 4	Walter Gander, Andrea Mazzario:	Numerische Prozeduren I
*	Nr. 5	Niklaus Wirth:	The Programming Language Pascal (Revised Report)
*	Nr. 6	C.A.R. Hoare, Niklaus Wirth:	An Axiomatic Definition of the Language Pascal
	Nr. 7	Andrea Mazzario, Luciano Molinari:	Numerische Prozeduren II
	Nr. 8	E. Engeler, E. Wiedmer, E. Zachos:	Ein Einblick in die Theorie der Berechnungen
*	Nr. 9	Hans-Peter Frei:	Computer Aided Instruction: The Author Language and the System THALES
	Nr. 10	K.V. Nori, U. Ammann, K. Jensen, H.H. Nägeli, Ch. Jacobi:	The Pascal 'P' Compiler: Implementation Notes (Revised Edition)
	Nr. 11	G.I. Ugron, F.R. Lüthi:	Das Informations-System ELSBETH
	Nr. 12	Niklaus Wirth:	PASCAL-S: A Subset and its Implementation
*	Nr. 13	U. Ammann:	Code Generation in a PASCAL Compiler
	Nr. 14	Karl Lieberherr:	Toward Feasible Solutions of NP-Complete Problems
	Nr. 15	E. Engeler:	Structural Relations between Programs and Problems
	Nr. 16	W. Bucher:	A contribution to solving large linear systems
	Nr. 17	Niklaus Wirth:	Programming languages: what to demand and how to assess them and Professor Cleverbyte's visit to heaven
*	Nr. 18	Niklaus Wirth:	MODULA: A language for modular multiprogramming
*	Nr. 19	Niklaus Wirth:	The use of MODULA and Design and Implementation of MODULA
	Nr. 20	Edwin Wiedmer:	Exaktes Rechnen mit reellen Zahlen
	Nr. 21	J. Nievergelt, H.P. Frei, et al.:	XS-0, a Self-explanatory School Computer
	Nr. 22	Peter Läuchli:	Ein Problem der ganzzahligen Approximation
	Nr. 23	Karl Bucher:	Automatisches Zeichnen von Diagrammen
	Nr. 24	Erwin Engeler:	Generalized Galois Theory and its Application to Complexity
	Nr. 25	Urs Ammann	Error Recovery in Recursive Descent Parsers and Run-time Storage Organization

* out of print