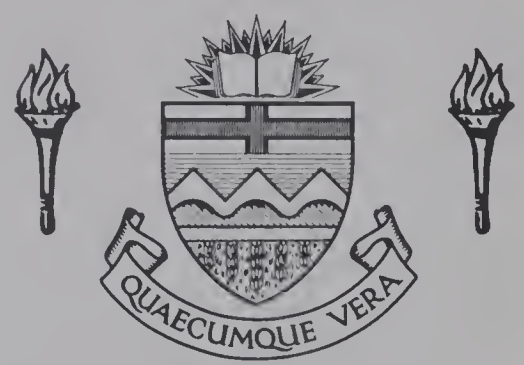


T H E U N I V E R S I T Y O F A L B E R T A

RELEASE FORM

NAME OF AUTHOR : Christopher Mark Thomson

TITLE OF THESIS : The Run-Time Structure of an ALGOL 68
Student Checkout Compiler

DEGREE FOR WHICH THIS THESIS WAS PRESENTED :
Master of Science

YEAR THIS DEGREE WAS GRANTED : 1976

THE UNIVERSITY OF ALBERTA

THE RUN-TIME STRUCTURE OF AN ALGOL 68 STUDENT CHECKOUT COMPILER

by

© CHRISTOPHER MARK THOMSON

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA

FALL, 1976

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, the thesis entitled THE RUN-TIME STRUCTURE OF AN ALGOL 68 STUDENT CHECKOUT COMPILER submitted by Christopher Mark Thomson in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

A run-time structure suitable for implementing a checkout compiler for ALGOL 68 is described. First, a set of design objectives are given; then the structures and algorithms used at run time are described. Difficulties with tracing are discussed. An outline is given of how dumping might be done. Finally, some pragmatic considerations are presented.

Acknowledgements

I would like to express my appreciation to Colin Broughton, author of the compiler portion of the FLASC system. his whip wielding, coddling, and pointed questioning were monumental in keeping me on the straight-and-narrow throughout the project.

I would also like to thank my supervisor, Barry Mailloux, for his support, and tolerance of my railing at aspects of the language which posed difficulties. Also, I would like to thank Jim Heifetz, who contributed many ideas in many long discussions.

Table of Contents

Chapter 1

Introduction

1.1 Design Goals

ALGOL 68 is, in many ways, a suitable language for teaching Computing Science. The language has well-defined syntax and semantics, and employs many of the concepts of Computing Science. It is also a "growth" language, in that a student can continue to use it as he becomes more sophisticated. This is not to say, however, that it is complete: there is still a need for other languages. However, there is no doubt that a student-oriented compiler system for ALGOL 68 is necessary to its acceptance as a tool for instruction.

In this thesis, such a system is described. The thesis concerns itself primarily with the design of the run-time system, or object-machine interpreter. The primary emphasis is on error checking, tracing and dumping, and how they are accomplished. The design described herein has been implemented (as a separate project) on an IBM /370 as the

FLASC system (Full Language ALGOL 68 Student Compiler, [4, 10]).

Many definitions have been given for the term "student compiler" [1, 3, 5, 6, 7], each differing slightly. Our design goals reflect what we mean by this term:

1. Fast compilation

   In a student "cafeteria" programming environment, the emphasis is on compilation: programs are compiled repeatedly until they appear to be correct, then are thrown away. For this reason, efficiency of execution is a strictly secondary consideration.

2. Extensive run-time error checking

   It is essential that checks be made for uninitialized values, subscripts out of range, scope violations, arithmetic overflows, and similar errors. All of these checks must be made at run time, since it cannot be guaranteed that a compile-time check will suffice in general.

3. Tracing and dumping

   It is important that the user be able to trace the flow of his program, as well as the values of key variables. Symbolic dumps are also of great use in discovering what has actually occurred in a

program run.

4. Lucid error messages

   Nothing is less informative than a "something went wrong somewhere" message. Care must be taken to ensure that error messages both locate and describe the error in a clear, comprehensible manner. It is often advisable to give typical causes and solutions.

5. Indestructability

   The system must be secure, in that the user must be restricted to his work space.

6. Cost limitation

   There must be provisions for imposing time and output limits on student runs.

7. Memory residency

   The use of overlays and utility datasets tends both to increase cost and to degrade real-time performance.

8. No object modules

   The compiler/run-time system interface is much simpler if object modules are not produced. This simplicity is reflected in lower cost of compilation. Independent compilation is generally unnecessary in a student facility, and is properly the domain of production compilers. This does

not, however, preclude this compiler from processing object modules from other compilers (of course, this violates security).

Many of the techniques described in this thesis are very time- and space-consuming. Some can be done more cheaply, but most cannot be improved by more than a factor of about two, which would have little impact upon the running time of a typical program. Program size is not considered to be very important.

1.2 Aspects of the Language

At the outset of the project, the decision was made to implement as nearly as possible the full language ALGOL 68, as described in [11], hereafter referred to as "the report". There were several reasons for this: (a) it was considered desirable to have a full-language implementation (as opposed to yet another subset); (b) the language described in the report has been carefully checked for ambiguities, and these have been removed; (c) a final authority exists for appeals about the meaning of obscure constructs; and perhaps most importantly, (d) no effort had to be expended in the design of the language to be implemented; rather, design of the implementation could begin immediately.

During the course of the implementation, some problems were encountered in the language, almost all in the

area of transput. This is primarily because, unlike the rest of the language, transput is not at all well described (the method of description being a program), and is riddled with errors. A few deviations were made to enhance the human-engineering aspects of the system. One aspect of the handling of loops may be considered different from the report's definition. This is discussed later.

There are several aspects of the language which other implementations have generally excluded, but which have been implemented in FLASC: parallel processing, flexible rows, and unions. Parallel processing is usually omitted because it precludes the use of a traditional Algol 60 stack. Flexible rows are often omitted because they also cannot be done in a stack model. Unions are usually omitted because they complicate the object code. Because the FLASC system implements all of these, it requires some nonstandard data structures to support them. These data structures are outlined in Chapter Two.

Of course, the primary purpose of a checkout compiler is to discover errors. Let us consider some of the types of errors that can be made in an ALGOL 68 program. Most obvious are syntax errors. These are not considered in this thesis, which is concerned with run-time errors only. Errors can be made in the formation or use of modes: these can all be detected at compile time. Tags can be used without being declared. This can usually (but not always!) be detected at

compile time. Attempts can be made to dereference names which have never been assigned to, or which are nil. This can be detected only at run time. Other errors which must be checked for at run time include: out-of-range subscripts, arithmetic overflows, transput errors, scope violations, memory overflows, nonterminating loops, runaway recursion, deadlock of parallel processes, assertions that do not hold, and arguments out of range for standard operators and procedures. Chapter Three describes these errors in more detail, and outlines their handling in the FLASC system.

One other important function of a checkout compiler is to aid the user in tracing the flow of his program, and, in the event of an error, dumping the values of variables. There are many aspects of ALGOL 68 which make dumping difficult and tracing ineffective. Chapter Four deals with these difficulties and some possible solutions to them.

Chapter Five discusses some of the pragmatic considerations of the FLASC system.

Chapter 2

Run-Time Structure

2.1 Type of Object Machine

The first and most important decision to be made in the design of an object machine is its basic nature; i.e., whether it is to be a stack, accumulator, or general register machine. This decision pervades the rest of the design.

In the FLASC system, a form of stack machine was chosen. There are several reasons for this choice. The most important is that compilation is greatly simplified, code generation being essentially a traversal of the parse tree. All the problems associated with register and temporary storage location allocation are thus avoided. Somewhat less important is that the treatment of values at run time is completely uniform; operands are always found in a standard order at the top of the work stack, and all results are left there. The stack machine has one important drawback, however: execution is very slow (especially on a

/370). This is considered to be much less important than the speed of compilation. One ramification of using a stack model is that support of garbage collection is quite costly. This is because the garbage collector must be aware of what is stored in the work stack during all phases of computations, since a heap generator may be used during a calculation; this means that the only pointer to the generated object is in the work stack. This is the only situation where the work stack contents need be considered. There are several methods of coping with this need: using self-identifying data structures, keeping a separate stack containing the modes of all the objects on the work stack, or keeping a separate stack containing all pointers inside objects on the work stack. The last method was chosen in FLASC, primarily because it is fastest. Note, however, that this method may preclude a compressing garbage collector, if (as in FLASC), only significant, rather than all, pointers are kept. Significant pointers are discussed later, after memory allocation has been described.

2.2 Type of Object Code

Use of a stack model specifies a great deal about the object code, but two more major decisions must be made: whether to generate standalone or threaded code [2], and whether or not to generate object modules. In FLASC, both decisions were made with simplicity of compilation in mind: threaded code, no object modules. This implies that the

code is generated directly into memory, and is not relocated after compilation. Thus the entire language processor (LP, by which both the compiler and run-time system are intended) is resident at all times. This consumes a great deal of space, but has the advantage that, since the LP is reusable, no part of it need be reloaded between runs. More importantly, the generated code can call directly those parts of the run-time system (RTS) which are needed. There is no need to "link edit" the generated code with the RTS.

Threaded code is a series of subroutine calls, interspersed with inline constants. On the /370, the calls are BAL instructions, which provide a means of accessing the inline constants. For example, the call to add two integers already on the work stack would appear as:

```
BAL   RET,XINTADD
DC    AL2(line,column)
```

The second word (two halfwords) is the source-listing coordinate: this is provided for the error processor, in the event of overflow. Note that when XINTADD is entered, RET points at the coordinate. To exit, XINTADD branches to offset four from RET. This is the address of the next BAL in the code sequence. Use of this scheme implies two important attributes of the RTS: at least part of it must be addressable from the generated code (i.e., there must be at least one base register pointing at the RTS), and the RTS will be essentially a large collection of subroutines, most of them quite small. Of course, not all of the RTS can be

directly addressable; even if all fifteen available registers were used, this would limit its size to 60K bytes, and not leave any work registers! Instead, a compromise was reached: three registers are set aside for base registers, and a special routine was written which calls other routines not normally addressable. Small, often-used routines such as integer addition are in the addressable portion, and large, seldom-used routines such as formatted input are in the portion not directly addressable.

One very important attribute of this threaded-code scheme is that (for generated code at least), the common /370 problems of addressability are completely avoided. This vastly simplifies code emission. Equally important is the fact that only offsets of entry points in the RTS need be known by the code emitter. This means that a much smaller number of relocations need be made when the LP is loaded, further reducing the cost of its use.

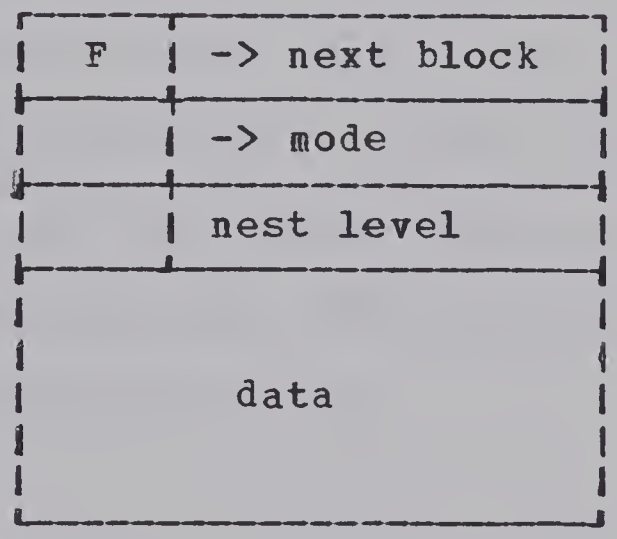
2.3 Memory Allocation

Memory allocation in FLASC is fairly simple. All memory is allocated in blocks which start with a standard "title". All blocks are allocated in the heap area. First-fit allocation is used, employing a roving pointer [9, pp. 437, 597]. First-fit is used because little (if anything) is known about the effectiveness (or lack thereof) of any other algorithm in an ALGOL 68 environment. It was chosen

for its speed and simplicity. Standard titles are used to simplify garbage collection.

Note that there is no Algol 60-type stack in this scheme. On the contrary, because of the needs of parallel processing, a cactus-stack arrangement is used. To make the stack model work, local stack frames (LSFs) are used, each of which contains the work stack area needed by the code generated for the range concerned. LSFs are described later.

Each block has two portions: a title and a data area. The title contains four parts: flags, including the



A Block

garbage collector marks and free/allocated bit; a "next" pointer used to chain blocks and also determine their sizes; a mode pointer which points at a tree used by the garbage collector to determine the form of the data area; and a nest level which is used in the scope check. Garbage collection and scope checking are described in later sections.

Local and heap cells differ primarily in the manner by which they are freed. Local objects are collected into stack frames, and freed explicitly when the range is exited, whereas heap objects are allocated individually, and freed when the garbage collector discovers they are no longer in use. Under this scheme, it is normal to call the memory allocator only once per range, to allocate its LSF.

2.4 Storage Structures

ALGOL 68, due to its complexity, requires many data structures at run time. In an effort to minimize complexity in the FLASC RTS, two goals were adopted: a minimum number of structures should be used, and the use of them should be uniform. Under this scheme, all refs, for example, look the same, regardless of what they refer to. This methodology simplifies all the algorithms which process the structures, especially the garbage collector.

Def Bits

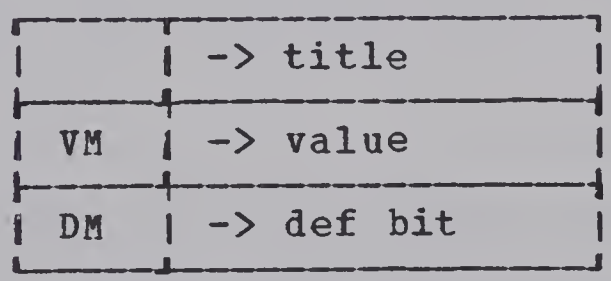
It is essential that a student LP check for the use of uninitialized variables. The FLASC system uses a special bit, the def bit, to determine the defined/undefined status of each cell. Because more than one name may refer to any one cell, def bits must be associated with values, not names, even though they are checked only when names are being dereferenced. Most, but not all, values have def

bits. Anything possessed by a tag has one. The only time an object does not have a def bit is when it is a structure or row (in which case the subobjects have them), or when it is known to be defined. The latter most commonly occurs in strings, which are assigned as units and thus normally must be well-defined. More on def bits later.

Def bits are checked whenever an object is moved onto the work stack. Most commonly this will be while dereferencing a name or pushing an object possessed by a tag onto the stack. Under no circumstances is an undefined value allowed on the work stack (skip is considered to be defined).

Refs

As mentioned above, refs have a standard form. They consist of a title pointer, which points to the title of the



A Name (ref)

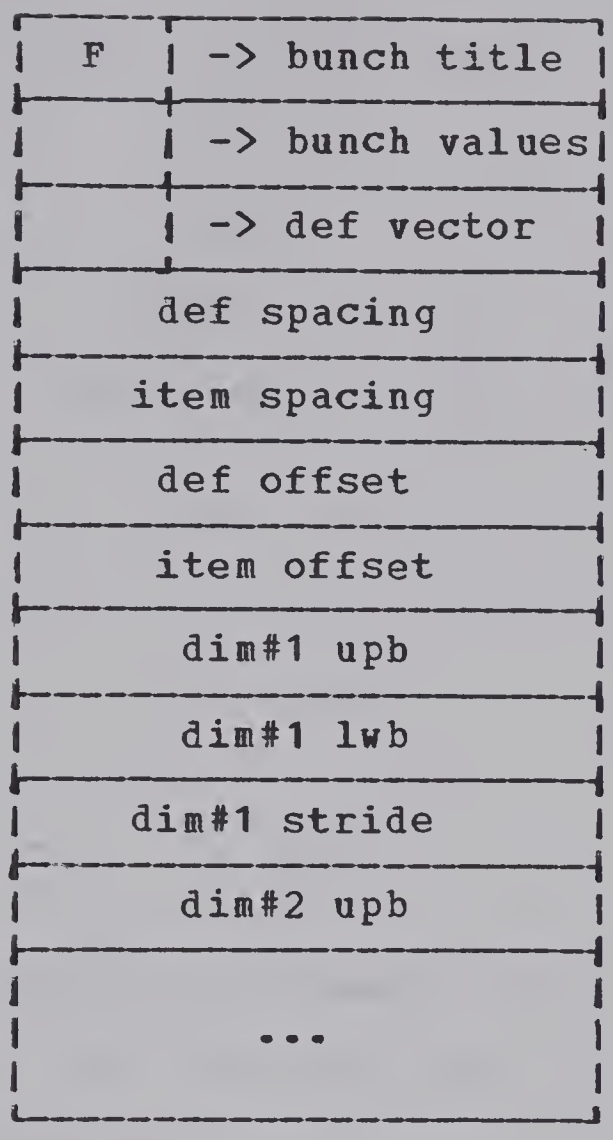
storage block containing the value; a value pointer; and a def bit pointer. If the value is a bool, then the VM field is used as a mask to tell which bit within the byte is used. Similarly, the DM field is a mask for the def bit.

The VM field could have been eliminated by storing bools one to a byte, thus achieving complete uniformity in addressing, but it was decided (and later regretted) that rows of bools should be packed, since these rows are typically huge, making the 8:1 space improvement desirable. Under the scheme used, bools are always treated as special cases.

Rows

Rows are represented by two data structures. One, of constant size (determined by the number of dimensions), is the descriptor, which is normally stored in the LSF. A flag field indicates whether there is really a bunch (sometimes there is no bunch). A bunch is the dynamic part of a row; bunches are described below. The descriptor also contains a bunch title pointer, for use by the garbage collector, a bunch value pointer, which points at the first element in the bunch, and a def vector pointer (which may be zero). Def bits of row elements are collected into a vector and stored in the bunch.

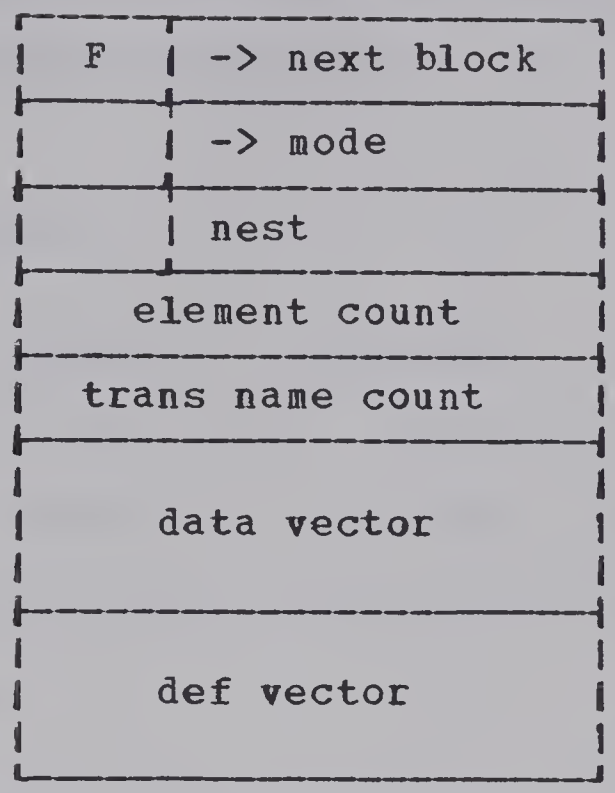
The remainder of the fields are used for slicing. Each dimension has a three-word descriptor, consisting of the upper and lower bound, and the stride. The stride is a multiplication factor used in indexing, and is the product of the "sizes" of the previous dimensions. It thus indicates the spacing of the elements of a given dimension. The item spacing is normally the size of each element in



A Row Descriptor

bits, but may increase during multiple selections. The def bit spacing is normally one, but may also increase during multiple selections. The item and def bit offsets are normally zero, but usually increase during slicing.

As mentioned above, a bunch is the dynamic part of a row. As such, its size can be determined only at run time. Thus, bunches are not put in LSFs, but are allocated at generation time in the heap area. Each bunch starts with a standard title. An element-count field tells the garbage collector how many items are in the bunch. The transient-name-count field is used for error checking during flexing



A Bunch

operations. The actual row elements are stored sequentially in the data vector, and their def bits (if any) are stored in the same order in the def vector. If the elements are stowed, or known to be defined, then there will be no def vector. If the row is flat, there will still be one element (even though the element count will be zero), which is used during bounds checking.

Structures

Structures are very simple. They are concatenations of their constituent fields, possibly in order of their alignment requirements, followed by def bits for the fields, in any order. If a field is in turn a structure, the field is treated as a separate structure; i.e., substructures are not broken apart to increase storage utilization. This

method leads to uniform treatment of structures, even when they are parts of other structures.

Miscellaneous Objects

A complex number is a structure of two fields, the real and the imaginary parts. Following these are the two def bits. Thus a complex is just like any other structure, although the RTS treats it in the same manner as an int or real in most cases.

A union consists of two fields. The first points at the mode of the current value, and the second contains the value. The second field is large enough to hold the longest of the possible values.

A procedure value has two fields. The first is a code pointer and the second contains the nest level (scope) of the routine.

A format is stored as a tree, in much the same manner as outlined in the report. A format value has two fields. The first points at the format tree and the second contains the nest level.

A semaphore is a structure of one field, a ref int, as suggested in the report. The int is allocated on the heap to avoid scope restrictions.

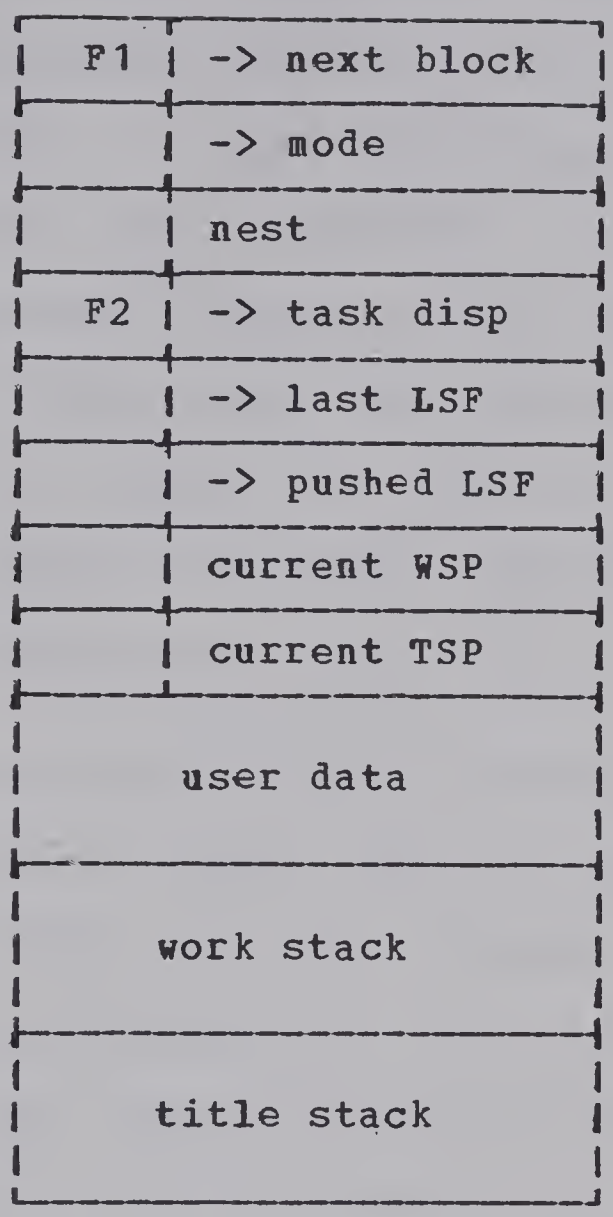
A channel is simply an int at run time, although the

user is not able to make use of this fact. A file is a structure with a single field, a pointer to an internal block. The def bit for the field indicates the open/close status of the file. The internal file block is much the same as that described in the report.

Tasks and Ranges

As previously mentioned, each range in the user program has a local stack frame (LSF) associated with it (provided it contains declarations other than loop control variables). LSFs consist of four main parts. The overhead portion is a standard title followed by a flags field, a pointer back to the task display (TD, described below), a pointer to the last (chronologically) LSF, a pointer to the last (recursively) LSF, and save areas for the work and title stack pointer registers. The user data area is next. It consists of all the local storage declared by the user, together with any def bits required (this area is just a structure). At the end are the work and title stacks.

LSFs are chained chronologically (via the last LSF field) for the benefit of range exit, return, and goto, which always process LSFs in reverse chronological order. These routines often make use of the mode field in the title to distinguish LSFs. A separate chain (the pushed LSF field) is used for recursion. All LSFs on the push chain are of the same type (i.e., belong to the same routine).



A Local Stack Frame

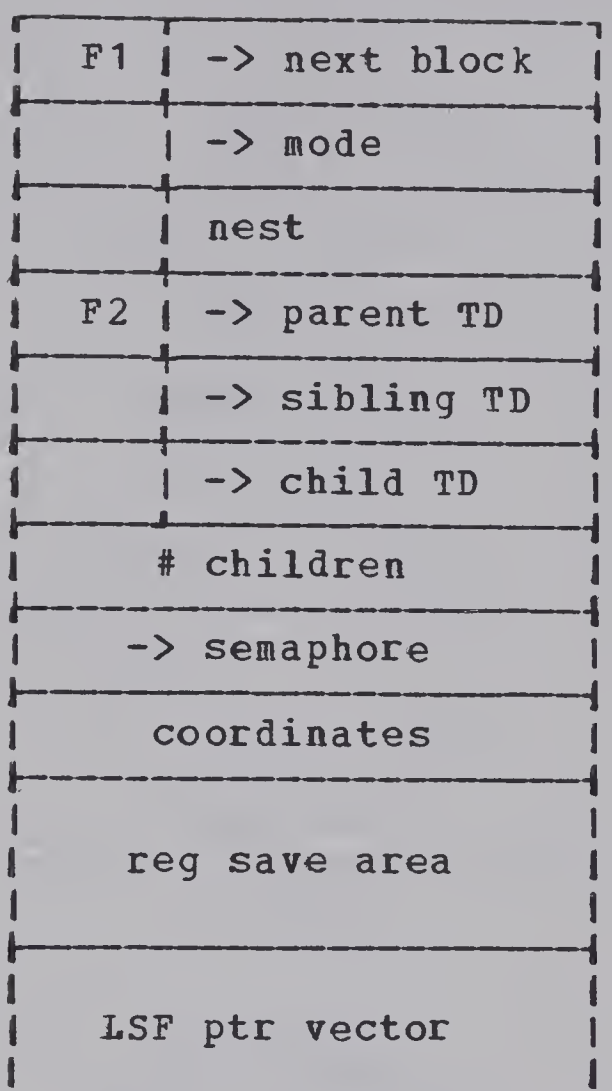
This chain is used to keep the task display accurate.

The work-stack area is used for the storage of all temporary and intermediate results. Its size is bounded and is determined at compile time by simple-minded "simulation" of the code generated. The amount of work space needed is bounded because elaboration of any construct which requires an unbounded amount of work space (e.g., recursion) causes a block entry, and therefore allocates a new work-stack area, the size of which can be calculated later (but which will also be bounded). The title stack is provided for the

benefit of the garbage collector. Each time an object containing a title pointer is pushed onto the work stack, this title pointer is also pushed onto the title stack. Thus title pointers are considered to be "significant" pointers (in the sense of Section 2.1), while value and def pointers are not. These stacks grow from the bottom of the diagram; that is, toward low memory. This facilitates access to values inside the stacks, since the /370 does not handle negative displacements very well.

Due to the block-structured nature of the language, values stored in outer ranges must be accessible. This requires some sort of task display (TD). Parallel processing requires multiple TDs. In FLASC, each task or process has a TD, which points to all the LSFs which are active in that task. All TDs have the same shape: creation of a new task merely involves replicating the old TD, then chaining appropriately. This ensures that all sibling processes start out with identical access to values. The initial TD is the "root" of all blocks in memory; the garbage collector starts here in determining what blocks are active.

A TD consists of two main parts: an overhead region and an LSF vector. The overhead region consists of a standard title; a flags field; pointers to the parent, next sibling, and eldest child TDs; a counter of the number of children alive; a pointer to any semaphore upon which the



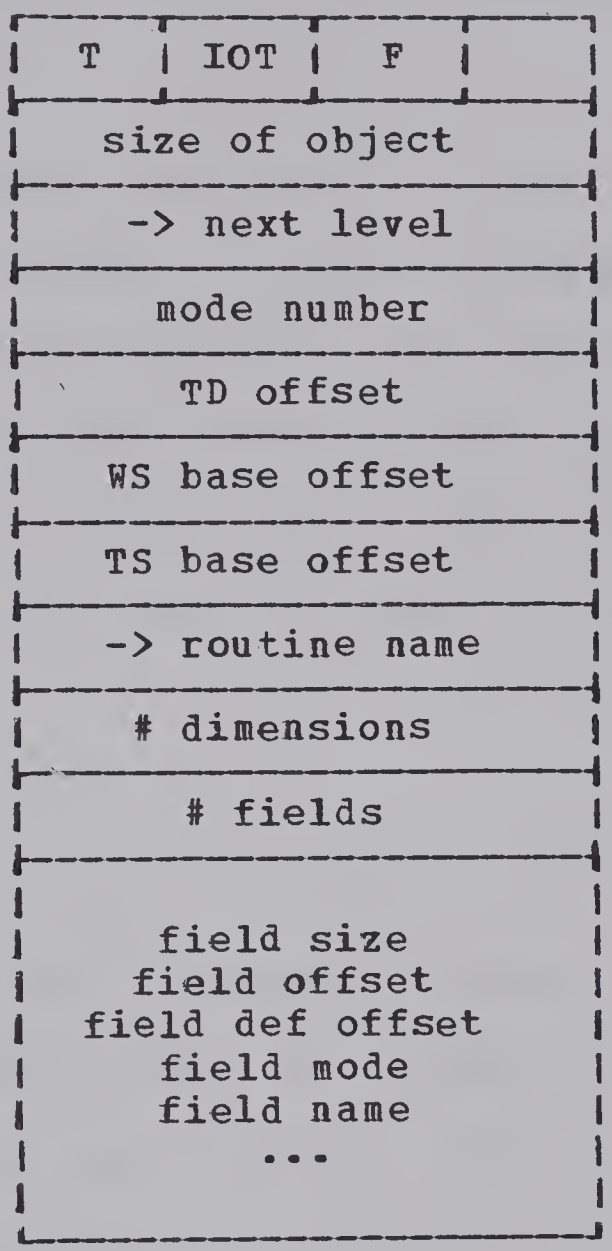
A Task Display

task may be waiting; the coordinates of the point of suspension (if any); and a register save area. The LSF vector has one entry for each type of LSF in the program. Each range which has an LSF has its own slot in the TD. The offset of any particular LSF is computed at compile time, and is used as an inline constant for primitives which access global values.

Mode Templates

Whenever it is necessary to know the mode of some object (such as a block headed by a title), a pointer to a

mode template is used. Mode templates are trees, each node



A Mode Template Node

of which is a word in the cannonical mode spelling. Structures and LSFs have nodes with field descriptors, causing multi-way branches in the trees. Other mode words, such as ref, result in unary nodes.

The T and IOT fields indicate the type of node; the F field is for flags. The mode-number and routine- and field-name fields are used for dumping. The TD-offset, and work- and title-stack (WS and TS) base-offset fields are used by

block entry. In the implementation, several of the fields are overlapped to save space, since a typical program will have many templates.

Mode templates are used by most utilities dealing with arbitrary objects. These include the garbage collector, dereference routine, most error checks, including defined, scope and bounds checks, generators, copy routines, transput, etc. In general, when a mode is not simple (e.g., real, ref int), a mode template pointer is given to describe it.

2.5 Algorithms

Most of the algorithms which massage the data structures outlined above are quite simple in nature, although some are not obvious. The algorithms described here can be broken into two broad classes: data-manipulation algorithms, and control-structure algorithms. The former are described first.

Denotations

Most denotations are handled by the compiler and require no run-time action other than copying them to the work stack. String denotations are somewhat different. The compiler passes a pointer and a length to the RTS and a descriptor is built on the stack. Note, however, that no bunch is allocated and the string is not copied. This is

one of the two cases when a row does not have a bunch.

Row and structure denotations (i.e., displays) are done by first elaborating the fields, then either enrowing or enstructing (described later). Note that in the case of structures, space is allocated in the work stack before elaborating the fields, to give "enstruct" a place to build the structure from the fields.

Dereferencing

Dereferencing is normally quite simple. For simple modes such as ref int, it consists of using the ref on the top of the work stack to address the value and def bit, check the def bit, then copy the value to the work stack, after popping off the ref. More complicateed modes require more effort. Often a mode template will have to be traversed looking for and checking the def bits. Copying the value to the work stack may not be sufficient; if the object points at any bunches or files, they, and any bunches they refer to, etc., must be replicated.

In an effort to speed up the RTS, special routines were written to handle the common modes such as int. Strings, in particular, are singled out for special treatment. Normally, dereferencing a string would involve copying the bunch, which would involve a memory allocation. If, however, the dereferenced value is destined to be used by a standard routine or operator, then no copy is made.

Because more than one dereference can be made at a single coordinate, the user is told the mode of the ref if there is an error (i.e., if the ref is nil or the value is undefined).

Assignment and Ascription

There are three kinds of assignment and two kinds of ascription. Assignment can be done to a row, a flex row, or a nonrow. Ascription can be done during a declaration or a call.

Assignment to a nonrow is simple, since a contiguous object is copied from the work stack to the area specified by the receiving ref. After the copy, the stack is popped and the def bit (if any) is turned on. Note that a scope check or bounds check will often have to be made, especially if a structure or a ref is involved. Also, if the value contains rows, then any bunches must be replicated. For reasons of efficiency, separate routines were written for each of the simple modes.

Assignment of a flex row is also simple. Here a check of the transient-name count must be made, in addition to the scope and bounds checks, but the operation is essentially a copy of the descriptor. Provision is made for row of plain, to avoid the scope and bounds checks and replications. This is primarily for strings (flex row of

char). Note that the row values never have def bits since they are on the work stack, and thus must be defined.

Assignment of a row is complicated. If the receiving row is a contiguous slice (i.e., the whole row, or some part in which all elements are adjacent in memory), then the assignment can proceed via a simple copy, after scope and bounds checks. If, however, the receiving row is a noncontiguous slice, then each receiving element must be indexed individually, then a copy made from the bunch being assigned. Note that in any event, the elements of the row being assigned are in order in memory, and can be copied simply, after any subbunches have been replicated.

Ascription is always a copy operation, the source being on either the current or the previous work stack (depending on whether in a declaration or call), and the destination in the current LSF. Both the work and title stacks are popped after the copy is made. Usually, the def bit associated with the receiving location is not turned on in the case of a declaration (see Chapter Three), but it is always turned on in the case of a call.

Slicing

There are several kinds of slicing: indexing, which causes the number of dimensions to decrease; subscripting, which yields a scalar; and trimming, which does not change the number of dimensions. Of course, all these can be done

on both ref rows and rows.

One of the first decisions made regarding slicing was that rows would not be operated on directly. Rather, they are "enreferenced", then sliced, then dereferenced. This reduces the amount of copying done, and simplifies the algorithms greatly.

Subscripting is probably the most common of slices, so an attempt was made to make it fast. In contrast to indexes and trims, which in the interest of simplicity are done one dimension at a time, subscripting is done all at once. That is, all the subscripts are pushed onto the work stack, and the addressing calculation is done in a tight loop. This may not seem significant until it is realized that indexing and trimming yield new descriptors, which must be allocated specially.

The subscript calculation proceeds according to the formula

```
acc +:= (index(i)-lwb(i))*stride(i)
```

The result of this calculation is the offset (in items) of the element within the bunch. The calculation of the actual addresses of the element and its def bit are done by multiplying this value by the appropriate spacing, and adding the offset, then converting from bits to bytes, and adding the value or def pointer.

Indexing is quite straightforward. The arguments are

the ref row, the index, and the bound number. A new descriptor is built in the heap area, having one less dimension than the old one. Next, all the dimension descriptors are copied, except the one being deleted. These remain unchanged because the spacing of the remaining elements in the bunch remains unchanged. The value and def offsets, however, are changed according to the formula

offset +:= (index-lwb)*stride * spacing

This is to account for the shift in the addressing caused by selecting an element other than the first (i.e., that selected by the lower bound) as the index for the operation.

There are four kinds of trims: renumber, triml (lower bound only), trimu (upper bound only), and trimb (both bounds). The algorithms for the last three are very similar. Renumbering (establishing a new lower bound) involves copying the descriptor and adjusting the upper and lower bounds. Offsets and strides remain unchanged, because the bunch is not affected. Trimu is also simple. It copies the descriptor, then adjusts the specified upper bound. Strides and offsets are not changed. Triml and trimb, because they change a lower bound, are more complicated. After copying the descriptor and changing the bound(s), new offsets are calculated according to the formula

offset +:= (newlwb-oldlwb)*stride * spacing

This is to account for the shift in the addressing caused by selecting an element other than the first as the lower bound

of the operation. The strides are not changed. Note that trimu, triml, and trimb are all required to involve implicit renumbering (the new lwb is 1). This is done last, by simply adjusting the bounds. No other change is involved.

In an effort to speed things up, trims allocate new descriptors only when instructed to do so by the compiler. If a series of trims is being done, then only one descriptor need be allocated, and then reused in subsequent operations.

Of course, in all slicing operations, the indices are checked against the bounds to ensure correct specification.

Multiple Selection

Multiple selection is really more like slicing than selection, since both the argument and result are ref rows (again row values are enreferenced). Normally, a new descriptor is allocated by a multiple selection.

Only the item offset, and the def offset and spacing are affected during the selection. The item offset is increased by the offset of the field within the structure. The item spacing remains unchanged, because the items are still embedded within the structures (i.e., no copy is made). If there is no def bit associated with the field (i.e., it is stowed), then nothing else is done. If the field has a def bit, then the def offset (which must

previously have been zero, since the row elements were stowed and thus had no def bits) is set to the sum of the old item offset and the def bit offset within the structure. The def spacing is set equal to the item spacing (which is equal to the length of the structure).

Selection

There are two kinds of selections: ref selections and value selections. Here again, enreferencing could have been used, but it was decided (arbitrarily) to implement two different routines.

Ref selections are very straightforward; after checking for nil, the offset of the field is added to the value pointer in the ref, and the def offset of the field (if any) is added to the def pointer.

Value selections are a bit more complicated. Since the structure is on the work stack, the field must be copied out of the structure, then the work and title stacks popped, then the field pushed onto the work stack, and the title stack pushed (for the field titles). It was decided that a memory allocation was undesirable, so the field is copied onto a lower part of the work stack (i.e., the WS is pushed), then copied back after suitable adjustments are made to the WS pointer. This is one of several routines which use "extra" work-stack space.

Rowing and Enrowing

There are four kinds of rowing, and two kinds of enrowing (used in row displays). A value can be rowed to form a row of mode; a ref to mode can be rowed to form a ref to row of mode; a rows of mode can be rowed to form a row rows of mode; or a ref to rows of mode can be rowed to form a ref to row rows of mode. A collection of mode can be enrowed to row of mode; or a collection of rows of mode can be enrowed to row rows of mode.

In all types of rowing, there is one object on the top of the work stack at entry, and a row or ref to row there on exit. A new descriptor is always built. In the first case (mode to row of mode), a bunch is also built. In the second case, no bunch is built, since the rules for refs in the language require that no copy be done. This is one of the two cases when a row descriptor has the "no bunch" flag turned on. In the third case, the descriptor on the work stack is simply stretched by one dimension. The same is true in the fourth case, except that, since the descriptor is not on the stack, a new one must be allocated, and a copy made.

In both types of enrowing, there is a group of objects on the stack at entry, and a row descriptor there on exit. In both cases, new bunches are allocated, and copies made of all subelements. In the first case, the objects on the stack are copied sequentially into the bunch. In the

second case, each of the old bunches (i.e., each element within them) is copied into the new bunch.

In rowing, none of the strides, offsets or spacings is changed. The new stride is the largest of the old strides. In enrowing, the def spacing is set to one, the def offset to zero, the item spacing to the size of the objects, and the item offset to zero. The stride of the new dimension is the largest of the old strides times the number of objects copied from the stack. Any bunches created by rowing or enrowing are without def vectors, since all the elements within the bunches must be defined (they came off the work stack).

Declaration, Generation, and Skip

Declarations, generators and skips are either trivial or nearly impossible, depending on the mode involved. Plain modes are trivial. Stowed and united modes are difficult.

Declaring a variable of plain or complex mode consists only of building the ref (if any). No action is taken to generate a plain mode, since space is allocated in the LSF at compile time, and this is left as zeros. Generating plain skips consists simply of pushing some zeros onto the work stack, and, in the case of complex, turning on the def bits.

Generating a complicated value involves more work. If

the mode contains rows, then the bunches must be allocated. In order to do this, all structures must be traversed, looking for rows as fields, and all rows recursed, looking for subrows. For this purpose, as well as the handling of heap allocation, a generate routine is needed. If it is given a location for the value to be generated, then this space is used; otherwise, space is allocated in the heap for it. Only the top level of any mode (except rows) is so allocated; space for the entire value is obtained in one piece. Scope can be either local or primal, depending on another parameter given by the compiler. Aside from the initialization of row descriptors after allocating bunches, all areas are left as zeros. This includes all def bit locations, so values are initially undefined.

The declaration of a complicated value involves elaborating the bounds (if any), possibly replicating them for repeated fields or modes, then calling the generator, ascribing the ref which is returned, then popping the bounds (which are not popped by the generator, to allow for joined declarations), and turning on the def bits for the tags declared. The reason for turning on the def bits last is given in Chapter Three.

Elaborating the bounds of certain modes may require a call. In particular, modes whose actual bounds contain generators which involve the current mode recursively will cause the code generator to recurse indefinitely if the

actual bounds are not made into a routine and called. Note, however, that this does not solve all problems: the routine must have the same scope as the declaration, or bogus scope errors may arise. This implies a special kind of block entry which does not change the scope level.

Skips of complicated modes are done as generators, for simplicity. Since generators return refs, a dereferencing must occur. However, since skip is required to be defined, all the def bits within the value must be turned on. This requires a special routine. Note that it is also necessary to fill in some value for unions. To enable this, the compiler suggests some mood (via the mode field of the template) for the "make defined" routine to use. If this mode is a row, then a bunch must be allocated.

Control-Structure Algorithms

Control-structure algorithms are those which implement syntactic structures which imply transfer of control, or which manage stack frames (i.e., block entry, block exit).

Block Entry and Exit

Block entry allocates space for a new stack frame, and the nest level is set to that of the old LSF, plus one. This increment does not occur if this is a block created by

the compiler around an actual bound. The old LSF is chained to the new one, then the pointer in the task display is updated, after being saved in the new LSF (this is the push chain). Finally, the new work- and title-stack pointers are set up, after saving the old ones in the old LSF.

If there is a value to be yielded by the block, then the value is copied to the old work stack before the block is exited. There is a special routine to do this. Note that a scope check may be required here. If there is no value, then this routine is simply not called.

Calls and Returns

Calls are done in several stages. First, a return address and coordinate are put on the work stack; then the arguments are elaborated; then the call primary is elaborated. At this time the actual call takes place. Control is transfered to the address given by the call primary (proc value). At the start of the proc, a block entry is made; then the parameters are ascribed from the old work stack. At this point the call is complete. Note that the arguments have been popped off the old stack, and all that remains there is the return address and coordinate.

Before returning, if a value is to be yielded, it is copied onto the old work stack, and popped off the current one. A scope check must be made here. The value copied goes above the return address on the old stack. Because of

this, a different routine is used from that which copies values at block exit.

When the value (if any) has been copied, a block exit is done. Now the actual return takes place. Control is transferred to the return address, and the address and coordinate are popped off the work stack. The returned value, if any, is at the top of the stack.

The coordinate of the call is pushed for the benefit of the traceback routine. In the event of an error, a trace of work stacks is used to find the coordinates of any calls. In order to distinguish calls from ordinary block entries, a flag is turned on in the LSF during a call, and turned off during the return.

Loops

There are two major kinds of loop: those with and those without control ints. Loops without control ints (loops with no for, from, by or to part) consist of the elaboration of the while part (if any), followed by a conditional jump to the exit (omitted if no while part), the loop body, then an unconditional branch back to the top of the loop.

Loops with control ints are of two types: those with to parts, and those without them. In either case, the from and by parts are elaborated (they may be implicit and

supplied by the compiler), and initialization of the control int takes place. Next, the to part (if any) is elaborated, and the test of the control int is made, provided there is a to part. Now the while part is done, followed by the loop body. At the end of the loop, the increment is performed. If there is a to part, then overflow is not an error, and if one occurs while incrementing the control int, the values of the control, by, and to ints are juggled to cause the test to fail. This will happen if "TO maxint" is given. If there is no to part, then overflow causes an error. In any event, the control int is updated, and a branch is made to the test at the top of the loop.

Choice Clauses

Boolean choices (if clauses) are very straightforward. The boolean expression is elaborated, and a conditional branch is made around the then part. There is an unconditional branch at the end of the then part, if there is an else part. The else part, if any, follows.

Integral choices (case clauses) are also quite simple. The integral expression is elaborated, and checked for being in range. If it is in range, then a pointer is selected from the branch table, and the appropriate clause is invoked. If the int is out of range, the out branch is taken. If the clause has no out part, the compiler supplies one. Following the branch table are the cases, each (except

the last) followed by an unconditional branch around the remaining cases.

United choices (case conformity clauses) are more complicated. First, the current mood of the chooser is found, then a search is made of the branch table to see which part (in or out) should be chosen. If no match is found, then the stack is popped, and the out part is taken. If a match is found, then the stack is compressed (the value required may be smaller than the union which contained it), and the selected in part is taken. If the mode required by the in part is a union, then the value must be reunited (the old union cannot be used, since it may be of a different size). As in the integral choice, each part (in or out) is followed by an unconditional jump to the end of the clause. If there is no out part, the compiler provides one, unless all the moods in the union have been mentioned in the in parts.

Parallelism

Parallel processes are handled as coroutines. Only one process is ever running at any time; all others are then waiting for service. Each process (task) has a task display, and these displays are chained together to form a tree. Scheduling is very simple: the task tree is traversed looking for some process which is ready to run (i.e., which has no live children, and is not waiting on a semaphore).

The first process found is started. If no process is found, the program has deadlocked. No effort is made at either deadlock prevention or aggrevation. A process gives up the CPU only when it "down"s a semaphore and is thus required to wait, or when it creates children via a parallel clause. More elaborate schemes could be used for scheduling, perhaps even timeslicing, but the cost would be very high, and everyone would have to pay, not just those using parallelism.

When a new task is created, the old task display is replicated, and chained. All stack frame pointers remain unaltered, since the new process is allowed to access all values global to the creating one. Note that since several processes are always created together (a parallel clause must have more than one unit), each process will initially have a sibling. The old process is made inactive by making the number of active children nonzero, and storing the registers and coordinates. The new processes have their registers initialized to the same values as in the old process. This implies that in the new processes, all the LSF pointers will point at the same LSF. No harm arises from this, however, because every new process allocates a new LSF immediately, so the same work- and title-stack pointers are stored repeatedly in the old LSF.

When a process terminates, it simply unchains its task display from the others in the task tree, decrements

its parent's child count, and calls the scheduler. Processes terminate only at the ends of parallel clauses or during gotos.

Gotos

At first sight, gotos may seem ghastly and impossible to implement, since they can jump out of an indefinite number of blocks, and an indefinite depth of recursion, as well as terminating an indefinite number of processes and transput operations. However, with the structures outlined here, the goto is very simple to implement. The compiler gives the branch address, and the mode of the LSF belonging to the range containing the label. The goto routine then loops, searching the LSF chain for a stack frame of the correct mode. Each LSF of the incorrect mode is exited. When the correct LSF is found, the pointer from it to its task display is followed, and the children fields of this task display are zeroed. Thus all stack frames are properly exited, and all processes terminated. The task displays and LSFs associated with the terminated processes are later garbage collected. Any transput operations in progress during the jump are implicitly shut down, since event routines are called using the same conventions as any other call, and all transput routines expect the case where no return is made.

This simple implementation of goto can be seen as a

clear case of good triumphing over evil.

A special case of a goto occurs with the label "stop". Stop is not handled as a label, but rather as a special routine, since if the user were to invoke it in the middle of his program, it would be highly undesirable to throw away all the information which might appear in a dump by exiting all the blocks in the program. Termination occurs immediately, and in the block containing the applied occurrence of stop.

2.6 The Garbage Collector

The garbage collector is a standard noncompressing mark-and-free garbage collector. The first stage marks all blocks (titles) which can be reached from the program, and the second stage passes sequentially through memory, turning off the marks and consolidating those areas not marked into the free list. The garbage collector is invoked only when an attempt to allocate a block of memory fails. If insufficient space is collected to satisfy the request causing the collect, the program is terminated. Note that runaway recursion will cause this type of termination.

The marking algorithm is a limited-stack, hard/soft-mark scheme, carefully designed to accept storage structures of arbitrary size and complexity. Initially, standard recursion is used (employing a pre-allocated, fixed-size stack), and each block encountered is hard marked. Whenever

a hard-marked block is encountered, a return is made immediately. If the stack overflows, then a pointer to the block is added to a secondary (and much smaller, pre-allocated) stack, so that marking can be resumed there later. A return is made from the overflow as if marking had continued normally beyond that point. Under this scheme, one phase of marking will mark the tree (or graph) up to a certain depth from the start node, and pointers will be kept to unmarked nodes at that depth. Thus, when the primary stack is about to underflow (signifying that marking is complete), a pointer is removed from the secondary stack, and marking is resumed. If that stack is empty, marking is finished, provided there are no soft marks.

Soft marks occur when the secondary stack overflows. The block causing the overflow is soft marked, and a count is updated. Then, marking is continued as though all were well, but when the marking would normally be complete (both stacks empty), the count is checked. If nonzero, then a scan of memory is made to locate a soft-marked block, and marking resumes at this block. If by some stroke of providence a soft mark is encountered during normal marking, it is made hard, and the count is decremented. Marking is complete when the stacks are both empty, and the soft-mark count is zero.

The occurrence of a soft mark is clearly a disaster. This is the reason for having the second stack. It is

believed that while chaining of blocks in an ALGOL 68 program may often be deep, it will rarely be both wide and deep. That is, there will rarely be more than a few (perhaps five) deep chains in a program. For this reason, a small secondary stack (say, thirty-two entries) should be sufficient to ensure that only the user testing the garbage collector will ever cause it to generate a soft mark.

Marking begins with the initial task display. Under normal circumstances, all blocks can be reached from there. However, under some circumstances the general copy routine causes a disconnection in the program tree, so a special mark pass may be required if the copy routine was active when the garbage collect was initiated. Similarly, the user may have associated texts with files which have no other pointers to them, so marking must be done on all file texts.

When marking is complete, a pass is made along the block chain and all marks turned off. If a block is found which is not marked, or which has not been flagged as a system block (e.g., file, book, profile block), then it is marked as being free, and merged with any free neighbours. Note is made of the largest free block so found, to see whether allocation will be successful.

Implicit in the above algorithm is that blocks are being collected, not words or bytes. This is because the principal entity handled by the garbage collector is a

title. Thus if a row is allocated, and only one element can still be reached, the entire bunch (and anything pointed at by refs in it) will be saved. This is considered too rare an event to be concerned about. More important is the fact that only title information is kept about the contents of the work stack. That is, when an object is put on the work stack (so that the collector need know about it --- it may be the only pointer to some other object), only its embedded title pointers are noted for the garbage collector (by putting them on the title stack, which is used during marking to handle temporaries). This precludes having a compressing garbage collector, since not all pointers are being considered.

There were several reasons for not having a compressing garbage collector. First, and probably foremost, compressing garbage collectors are very complex and nearly impossible to debug (and they always have bugs --- the SPITBOL garbage collector still had bugs three years after distribution, in spite of several attempts by the authors to clean it up). Second, they are slower, and the gain in memory utilization does not appear to be great (first fit is almost always over ninety-five percent effective in its use of memory [9, pp. 447-450]). Third, they require complete knowledge of any temporaries, which, in this case, would either mean pushing all pointers of any type, or pushing the modes of objects on the work stack. This, however, would mean that both stacks would be pushed

for modes like int, which is much too costly.

The only apparent advantage in having a compressing garbage collector is that memory allocation is very simple: typically a subtraction from a pointer. However, since there is little reason to believe that memory allocation is a bottleneck, the extra cost of allocating by block is deemed to be acceptable.

Chapter 3

Error Checking

3.1 Types of Errors

There are some sixty types of error which can occur at run time in ALGOL 68. This chapter describes some of these, and the methods of detection used in FLASC. Unfortunately, there are whole classes of errors which are not detected by FLASC. These are discussed later.

The most common run-time errors include: arithmetic overflows, undefined values, division by zero, undeclared tags (this really is a run-time error; more on this later), attempts to slice, dereference, assign to or select from nil, memory overflows, scope violations, deadlocks of parallel processes, subscripts out of range, bounds which do not match (during assignment), indefinite loops or recursion, assertions which do not hold, page or line limits exceeded, invalid characters in input, several dozen other transput errors, and arguments of standard functions out of range.

To gain an appreciation for just how many error checks need be made in a typical program, consider the following clause:

```
a := b + c;
```

'a', 'b', and 'c' are all ref ints. How many checks must be performed here?

Clearly, an overflow check must be made for the addition. Also, it is apparent that undefined value checks must be made on 'b' and 'c'. It is less obvious that 'a', 'b', or 'c' might be nil. (They may have been ref int parameters, and nil could have been passed in.) Least evident is the fact that 'a', 'b', or 'c' may never have been declared, since declarations may have been skipped (see next example). Thus this innocuous-looking clause requires nine checks, even though most production compilers would produce only three machine instructions!

Because declarations are elaborated at run time, and because units used for initialization may contain gotos, it is possible to jump over some declarations:

```
BOOL f;
read(f);
BEGIN
     INT i := IF f THEN 1 ELSE GOTO a FI;
     INT j := 2;
  a: print(i+j)
END
```

'j' is declared if and only if 'f' is 'true'. This can be

determined only at run time. Thus, in a naive compiler, the occurrence of a tag must involve a run-time check to determine whether it is declared, even though it appears in a declaration! This kind of problem need not involve a goto: ordering of declarations is also important; for example, in

```
REAL a := p(2);
PROC (REAL) REAL p = sin;
```

'p' is unknown when 'a' is to be initialized. Note that the identification rules of ALGOL 68 require that the inner 'p' be identified, rather than some outer one.

In a similar vein, consider:

```
INT a:=3, b:=a+1, c:=b*2;
```

Here, 'a' and 'b' are being used before they are guaranteed to be defined, because of the rules of collateral elaboration.

Now, also with respect to collateral elaboration, consider the following example:

```
STRING s := "abc";
s[3] := (s := "ab")[3];
```

If the right-hand side of the second assignment is elaborated before the left-hand side, then a subscript error occurs. If the left-hand side is elaborated before the right-hand side, then an attempt will be made to flex 's'

while there is a transient name outstanding (on the work stack).

Perhaps the ugliest of all checks is the scope check, which ensures that no ref can be made to refer to a value which will "go away" before the ref does (i.e., no refs can be left pointing off into outer space). Perhaps the simplest example which demonstrates that this check must be done at run time is the following:

```
PROC copy = (REF INT i) REF INT : i;
REF INT j;
BEGIN
     INT k;
     j := copy(k)
END
```

Here, 'copy' could have been made arbitrarily complex, without changing its function, so that no compiler could detect the scope violation in the fifth line. This means that the check must be done at run time. The check is often far from simple, though, as the following shows:

```
[3] REF INT ii :=
     BEGIN
          [3] REF INT jj := (NIL, NIL, NIL);
          INT k;
          BOOL f; read(f);
          IF f THEN jj[2] := k FI;
          jj
     END;
```

Here, a row of ref int is being yielded by the 'begin' clause. The scope of this row will depend on the refs within it. In particular, it is initially of primal scope (and hence can be assigned to anything), because all its

constituent parts are nil (and thus primal). Now, depending on the run-time value of 'f', the scope is made local, by assigning 'k'. Thus, depending on 'f', there may be a scope error. But the only way to detect this is to break open the row, and check each element for invalid scope. This same sort of thing has to be done with structures.

An overflow error can arise in loops:

```
FOR i FROM maxint-2 DO
     print(i)
OD
```

Here, 'i' will clearly overflow on the fourth iteration. The only reason this is remarkable is that the report could be interpreted to state that the loop should continue past the fourth iteration, but gives no clue what to print. It is also important to consider the inverse situation:

```
FOR i FROM maxint-2 TO maxint DO
     print(i)
OD
```

Here, no overflow should occur, and the loop should terminate without incident after three iterations. However, if the algorithm given in the report is followed, an overflow will occur during the increment at the end of the third iteration. This was the special case mentioned in Chapter Two.

Sadly, there is a very large class of very common errors which are not detected by FLASC, and, indeed, cannot

be detected by any LP. These involve constructs which are undefined due to the rules of collaterality. These rules state that the order of elaboration of the two sides of assignations and operations, the arguments of calls and slices, and a host of other things is left undefined. Thus, any program which could yield different results with different "legal" orders of elaboration is undefined. Here are some examples of this phenomemon:

```
INT i := 1;
PROC inc = (REF INT a) INT : a +:= 1;
i + inc(i);                          #boom#
(i := 2) + i;                        #boom#
STRING s := "abc";
s[1] + (s := "ab");                  #boom#
s[inc(i)] := s[i];                   #boom#
[6,6] INT j;
j[inc(i),inc(i)] := 1;               #boom#
```

The first two examples are probably the simplest. Here, depending on which side is done first, different results will occur. The third involves a more subtle problem. One of the valid orders is to perform an index, then go do the other side, then come back and dereference. This results in an attempt to flex while there is a transient name outstanding. The fourth will duplicate one of the characters within the string, but which one? The fifth could select different elements of the array.

3.2 Some Error-Checking Techniques

Many of the error checks are very straightforward. For example, subscript checking is done during slicing,

since the descriptors have the necessary information available.

Checks for nil are very simple: when a name is required which cannot be nil, a check for a zero value pointer field is made (nil is always zero).

Most arithmetic errors (such as overflow, underflow, divide by zero, etc.) are handled as program interrupts, which are caught by the operating system interface. The method used to tell the RTS that an interrupt has occurred is via a BPI (branch on program interrupt) "instruction". This is a noop which follows any instruction expected to interrupt, specifying a branch address and interrupt type. If an interrupt of the specified type occurs, then the branch is taken. This allows the RTS to recover from interrupts in a very controlled manner. Note that not all interrupts result in an error. Overflow during a loop increment (if the loop has a to part) is not an error.

Memory overflows, as mentioned earlier, are caught by the garbage collector, after it tries and fails to recover sufficient space to satisfy the current request for memory. Stack overflow (i.e., runaway recursion) is caught in the same way.

Deadlock is detected by the scheduler, when it cannot find a ready task to dispatch.

Output limit overruns are caught by the newline and

newpage routines. This check is made only on the standard output file. Time limit overruns are caught by the operating system interface, and a global flag is set. The RTS checks this flag prior to executing any function which might result in a loop. This includes gotos, loop bottoms, and calls.

Assertions, which are handled by the ASSERT operator, are trivial to check.

The checks which pose difficulties are the undefined, scope, bounds, and transient-name checks. These are now described in some detail.

Undefined-value checking is the most pervasive and expensive check made. It is estimated that as much as 25% of the run time is spent doing undefined-value checking. By way of justification, this is also the most common error made by students.

It was considered essential that an accurate way be found to perform the undefined-value check. By accurate, it is intended that all programs containing errors be stopped, and that no valid programs be stopped. This criterion rules out the method of setting aside a special value as undefined (besides, it is not clear what value of bool or char to make undefined). It should be noted that both WATFIV [6] and PL/C [5] use this method, and in both it is possible to cause erroneous terminations. For example, in the WATFIV

program,

```
      INTEGER I, J, K
      INTEGER M/Z01010101/
      DO 10 I=1,256
      J = (I-1)*M
      K = J
   10 PRINT 1, K
    1 FORMAT(' ',A4)
      STOP
      END
```

the message "J UNDEFINED IN LINE 5" is given, even though this is patently untrue. The simplest way to solve this problem is to use a separate bit (a def bit) to give the defined status of the value. Having decided to use an "extra" bit, it must next be decided where this bit should be put. Two possibilities arise: the bit can be put at some fixed point with respect to the value (e.g., at the beginning or end), or it can be put at some arbitrary location, unrelated to the location of the value. The former has the advantage of higher speed and smaller names, while the latter has the advantage of better storage utilization, especially in arrays. It should also be noted that unless the def bit is put at the front of the value, neither def bits nor values are handled in a uniform manner, which violates a previous design goal. It was decided on the basis of better packing in rows, and the uniform treatment, to make the def bit separate from the value.

As some of the examples in the last section showed, there are ways other than a lack of initialization to give rise to an undefined value. In FLASC, all these situations

are taken care of by associating a def bit with the entity in question. In particular, each tag has a def bit (as does any value it may refer to). This is regardless of what mode the tag may be. Thus, whenever a tag is encountered, its def bit is checked to see if the tag has been declared. In order to catch collaterality errors in declarations, the def bits for the tags being declared are turned on at the very end, so that if the tags appear in initializations, an error will result.

Not all def checks are simple and straightforward. Only those cases where the mode is not stowed lead to simple solution (i.e., simply checking the bit pointed at by the ref). Stowed modes have multiple def bits, which may be difficult to address. In particular, a structure is not required to have the def bits of its fields contiguous and starting on a nice boundary, so each must be addressed separately. The addresses of the bits can, of course, be determined from the mode template, but this is a slow process.

Rows may or may not have fast def checks. If the def vector pointer is zero, then there is no check at all, since this condition assures that the entire row is defined. If there is a def vector, then chances are that it is contiguous (this happens when the elements of the row are contiguous). If so, then a trick can be used to check all the bits at once. This is done by use of the COMPARE LONG

instruction of the /370. A pad character of X'FF' is used in conjunction with a length of zero to check all the def bits in all but the first and last bytes. These must be handled separately.

If, however, the row is not contiguous, and so the def bits are not contiguous, then each def bit must be addressed separately. This is very slow. This situation can arise when a ref row has been sliced. To implement this addressing, a work area is needed to store the indices. This gives rise to one of the few implementation restrictions on the language accepted: in FLASC, rows can have only up to 255 dimensions. The question now arises where to put this work area. Fortunately, a ref must appear between any two noncontiguous rows in a path through a data structure (i.e., it is possible to have ncrow of ref to ncrow, but not to have ncrow of ncrow). This means that the work area for indices can be statically allocated, since the def check stops if it encounters a ref. (Incidentally, other recursive utilities have the same property, and use the same static work area.)

Of course this "extended" def check is recursive in the mode. That is, if the mode is struct of row of union of struct ..., then the routine will recurse. This implies a stack. This stack is also statically allocated, and implies another of the implementation restrictions of FLASC: rows, structures and unions can be nested to a maximum depth of

255 without an intervening ref. (Again, other recursive utilities use this stack.)

Scope checking corresponds rather closely to def checking. Here again, the mode must be recursed (in general), looking for all title pointers, to compare their scope with that of the receiving name. Again, if a ref is encountered, the check does not go below that level, since that ref must have the correct scope (it has been assigned, so a previous scope check was done).

Bounds checking must also do this recursion, comparing the bounds of the source rows with those of the destination rows (bounds checking is done only during assignment). There are two complications, though. If a particular level is flex, then the check is bypassed, but only for that level. Other levels must still be checked. If a particular level is flat, then not only must the bounds be identical at this level (1:0 does not match 2:0, even though both are flat), but checking must continue to levels below that which is flat. This means that even flat bunches must have at least one element, as mentioned in Chapter Two.

Even though many (most?) collaterality errors having to do with transient names will never be detected (because FLASC does things in a left-to-right order), some sort of check is required to maintain integrity of the data structures. This check is accomplished by having a transient-name count (TNC) associated with each bunch. This

count is updated whenever a transient name pointing into that bunch is created (via slicing), and downdated when the name is destroyed (via voiding or dereferencing). Assignment to a flex row checks this count, and gives an error if it is not zero.

This count updating is done via two routines: deflex and dectnc. Deflex takes a ref flex row, and returns a TNC pointer and a transient ref row (transient refs look the same as other refs). In the process, it increments the TNC. The TNC pointer is above the ref in the stack, and stays there though subsequent actions (e.g., slicing, rowing, etc.). When a transient ref is to be voided or dereferenced, dectnc is called. It chases the TNC pointer, decrements the TNC, and deletes the pointer from the stack.

Note that when a balance of ref and transient ref is made (yielding transient ref), a call is made to another routine, maketrans, which pushes a zero TNC pointer onto the stack. Therefore, dectnc must be prepared to accept a zero pointer. This situation will also occur if a nil ref flex row is rowed.

Chapter 4

Tracing and Dumping

Of course, not all errors have the good manners to make themselves known by causing an immediate and correct diagnostic message. As we all know, just the opposite is more commonly the case. For this reason, it is extremely desirable to have some sort of tracing and dumping facility. A clear case for this is made in [5], for the PL/C compiler. Tracing has also been implemented in SNOBOL [8] and SPITBOL [7]. These systems provide especially useful tracing facilities. Unfortunately, ALGOL 68 does not lend itself to tracing.

Dumping can often be as important a debugging aid as tracing. It has the advantage that it is only done once, as opposed to tracing, which tends to be a continuous, paper-wasting process. Dumps are an invaluable aid in determining what "really happened" in the program. However, unless the dump is symbolic, it is of little or no use to the student user. SPITBCL provides an excellent dumping facility.

Also of considerable diagnostic use is some sort of flow trace (which has also been implemented in PL/C), and, in some cases, an execution profile. In almost all cases, a flow trace of, say, the last 25 branches taken is sufficient. This is simple to provide. Gathering of profile information is also quite simple, and can prove invaluable in avoiding wasted effort speeding up seldom-used modules. This facility has been provided in ALGOL W [1].

4.1 Tracing

Unlike other languages which provide suitable tracing facilities (most notably SNOBOL), ALGOL 68 has the orthogonalized concept of a ref. This causes severe difficulties in tracing. In fact, it makes tracing ineffective.

In most languages, "variables" are traced. In ALGOL 68, there are no variables. Instead, tags possess refs, which point at values. This, in itself, causes no problems. However, when an argument is passed to a proc, it may be passed as a ref. For example, in

```
PROC p = (REF INT j) VOID : j+:=1;
INT i;
PR trace i PR
i := 1;
p(i)
```

'i' is passed as a ref. But how can we trace 'i'? Inside 'p', it is known as 'j', so any message should say "j = 2 in

line 1"; but then it is not clear that this 'j' is the same thing as 'i'. If, on the other hand, we say "i = 2 in line 1", massive confusion could result.

There is, however, a more fundamental problem involved here. 'p' need not be called with 'i' as its argument. How, then, do we decide whether or not to produce a trace message? This problem can be solved by associating a flag (like the def bit; a trace bit) with the value, which would be checked during assignment, and, if on, would result in a message. This technique will still not tell us whether to print 'i' or 'j', though (i.e., what printable designation to use).

This problem of printable designations is a serious one. There exist many objects which will never have them, e.g., most heap values. The problem is even worse when the user wishes to trace pointers (i.e., ref refs, or ref ref refs). Here, not only is it unclear what printable designation to use, but also what to print as a value.

One possible solution to this dilemma is to disallow tracing of any but "simple" tags (i.e., variables of plain modes). This will still not cope with ref parameters, however. There are two alternatives open: associate with the value not only a trace bit, but also a trace string, which is to be used as the printable designation during tracing; or make no attempt to trace the value in the proc, but instead produce a possibly spurious message upon return

from the proc (this need only be done for ref parameters). Neither method is really satisfactory. The first prints messages with incorrect names in them, while the second produces messages at the wrong time (as well as producing extra messages).

Still another difficulty is the multiple use of similar tags. It is very common to have several 'i's in a program, and a message saying only 'i' would be insufficient. It would also be difficult to tell the compiler how to trace the various 'i's, especially if they occur in nested blocks. One possible solution would be to print the coordinates of the declaration along with the message.

Due to these and other more esoteric difficulties, it was decided that tracing should not be attempted in FLASC. The philosophy behind this decision was that since tracing could not be guaranteed to work, it should not be included at all. It was felt that the user could perform much better and (to him) more meaningful tracing than the compiler, by using the equivalent of print calls. To make this a bit more palatable, a trace call (which looks and acts like print) has been included. This routine checks the value of a user-accessible boolean variable "trace flag", and prints messages only when it is true.

A particularly useful tracing feature in SNOBOL is function tracing, which gives a message each time a function

is entered or left. This message gives the function name, the level of nesting, and the arguments and result of the function. Unfortunately, all of the difficulties arising in value tracing recur here: the proc may not have a printable designation, and it may not be possible to print the values of the arguments or result in a meaningful way. This is very likely to be the case for arguments, since they would commonly be refs. Function tracing was therefore not attempted.

Branch tracing and profile gathering have, however, been included. These keep track of "major decision points". These are the various points in the program where control flow is altered from the sequential. Such points include the branches implicit in 'if', 'case', 'do' and parallel constructs, as well as the explicit ones in gotos, calls and returns. The branch trace dumps the last 25 major points passed, while the profile gives a count of how many times each major point has been passed. The profile is output as a bar graph, sorted by coordinate. Profile information is optional.

4.2 Dumping

Dumping, though more tractable than tracing, is still not simple. Here also, the concept of ref is diffcult to handle. It is possible (and not unreasonably difficult) to give a complete, absolutely accurate dump. However, this is

not likely to be what the user wants, especially not the first-year student user. Such a dump for the program given in the last section might look like:

```
stack frame : p
   j : #1
stack frame : main
   p : #2
   i : #3
primal environ

#1 (ref) : -> #4
#2 (proc) : procedure
#3 (ref) : -> #4
#4 (int) : 2
```

(It is possible in this case for the RTS to give the name 'p' to the first stack frame, although in some cases it could not give any name.) In this simple case, the dump may appear acceptable, but what happens with more complicated programs? In particular, in such a scheme, strings would be dumped as rows of characters. For example:

```
[3] CHAR s := "abc";
REF CHAR c = s[2];
INT i;
```

might produce:

```
stack frame
   s : #1
   c : #2
   i : #3

#1 (ref) : -> #4
#2 (ref) : -> #7
#3 (ref) : -> #5
#4 ([1:3]) :
   [1] : #6
   [2] : #7
   [3] : #8
#5 (int) : undefined
```

```
#6 (char) : "a"
#7 (char) : "b"
#8 (char) : "c"
```

Note that in general, rows and structures (including LSFs) have to be broken open in this manner, to display the subnames properly. It is suggested that the user would much rather see:

```
stack frame
   s = "abc"
   c = "b"
   i = undefined
```

even though this does not preserve some of the information about the refs in the program. Normally this detailed information is required only for refs to heap values (i.e., linked lists). In this case, the dump must necessarily assume a format similar to the first one given, since there will not normally be any printable designations to give to nodes in the lists. Rows, of course, always present a problem, since they are usually large, and must be displayed element-by-element.

Dumping in FLASC is still undergoing evolution, but currently the user can select between a full or partial dump, in simple or complete format. A partial dump omits objects not in LSFs (i.e., row bunches and heap values). Complete format is the first shown above; simple format is the last. Rows and structures are dumped element-by-element regardless of format, but in complete format, an extra level of indirection is given.

Chapter 5

Some Pragmatic Considerations

As no doubt became apparent in Chapter Two, the FLASC system was written in /370 assembler. The reasons for this are quite simple: all the alternatives examined either required writing and/or maintaining the version of the compiler in which FLASC was to be written, or was orders of magnitude too clumsy or inefficient to be used (PL/I was in this category). Both authors of FLASC had considerable prior experience in writing large assembler programs, so it was felt that few problems would arise due to poor understanding or programming practice. This has turned out to be the case.

The compiler and run-time system comprise about 40000 lines of code, and occupy about 120K bytes of memory. An additional 40K bytes are required for standard tables. A small program (<200 lines) will compile and run in less than 250K bytes.

Because it is in assembler, and because the /370

instructions MVCL, ICM, STCM and CLCL are so useful, FLASC will not run on a /360 without very costly operating system support to interpret these very commonly-used /370-specific instructions. FLASC is, however, operating system independent. It runs under MTS, OS/VS, and CP/CMS. Other operating systems can be accommodated by rewriting the operating system interface, which is about 2000 lines of code.

At the time of this writing, the compiler is nearing completion, and the RTS is complete except for some parts of formatted transput and dumping. Most of the RTS has been tested by running hand-coded programs.

Possibly the best way to ensure the doom of a student compiler is to produce obscure and inaccurate diagnostic messages. A great deal of careful thought has been put into the FLASC diagnostics, and it is felt that most of them are now adequate. This, however, cannot be verified until the system has actually been used by students. This will occur soon. When it is known what types of errors are most common, the diagnostics (and compile-time fixups) can and will be made much more effective.

A very important consideration concerning any student compiler is the cost of its use. In the tests performed, the run-time cost of FLASC compared favorably with that of PL/C, the only available system which could be considered comparable. It is expected that compilation costs will be

somewhat lower than those of PL/C.

Is it worth it? There was a severe and constant temptation to change the language to make the system both easier and more efficient, but this was not yielded to. Perhaps one of the most valuable lessons to come out of this effort is that a language should be designed with errors in mind. ALGOL 68 was not, and consequently it has many types of errors which are difficult to understand, detect, or recover from. Both the syntax and the semantics suffer from this. However, compared to the other three major general-purpose languages (FORTRAN, ALGOL 60, and PL/I), ALGOL 68 provides a flexibility and naturalness that makes it a nicer language to program in. It is therefore considered valuable to have a checkout compiler for this language.

References

[1] Bauer, H., Becker, S., Graham, S., "ALGOL W Implementation", Tech. Rep. CS98, Computer Science Dept., Stanford University, (May 1968).

[2] Bell, J.R., "Threaded Code", Comm. ACM 16, 6 (June 1973), pp. 370-372.

[3] Boulton, P.I.P., Jeanes, D.L., "The Structure and Performance of PLUTO, a Teaching-Oriented PL/I Compiler System", INFOR 10, 2 (June 1972), pp. 140-150.

[4] Broughton, C.G., Thomson, C.M., "Aspects of Implementing an ALGOL 68 Student Compiler", Proc. 1975 Internat. Conf. on ALGOL 68, Stillwater, Oklahoma, (June 1975), pp. 23-37.

[5] Conway, W.C., Wilcox, R.W., "Design and Implementation of a Diagnostic Compiler for PL/I", Comm. ACM 16, 3 (March 1973), pp. 169-179.

[6] Cress, P., Dirksen, P., Graham, J.W., FORTRAN IV with WATFOR and WATFIV, Prentice-Hall, Englewood Cliffs, New Jersey, 1970.

[7] Dewar, R.B.K., "SPITBOL Version 2.0", Illinois Institute of Technology, February 1971.

[8] Griswold, R.E., Poage, J.F., Polonsky, I.P., The SNOBOL4 Programming Language, Second Edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[9] Knuth, D.E., The Art of Computer Programming, Volume 1, Second Edition, Addison-Wesley, Reading, Massachusetts, 1973.

[10] Thomson, C.M., "Error Checking, Tracing, and Dumping in an ALGOL 68 Checkout Compiler", Proc. Fourth Internat. Conf. on the Implementation and Design of Algorithmic Languages, New York University, New York, (June 1976), (to appear).
Also in SIGPLAN Notices, to appear.

[11] van Wijngaarden, et al., "Revised Report on the Algorithmic Language ALGOL 68", Acta Informatica 5, 1-3, (January 1976).