Department of Computer Science
University of Cape Town

A Control and Sequencing Language

by Colin A. Fair

A Thesis
Prepared Under the Supervision of
Associate Professor G. D. Smit
In Fulfilment of the Requirement for the
Degree of Master of Science in
Computer Science

University of Cape Town

1991

Abstract

In a process control environment, batch processes, as opposed to continuous processes, are characterised by multi-product manufacturing lines which often involve frequent product changes. One component of batch control systems is a programming language which is used to control and synchronise the operations of the plant. Initially low-level languages (e.g. ladder logic, boolean algebra and assembly language) were used, but have now been replaced by specialised high-level languages. These languages provide more functionality and are easier to use. The dissertation examines one such high-level sequencing language (CASL) and identifies functionality, clarity and readability improvements that can be made to the language. An implementation of an upwardly compatible compiler for the improved language is described briefly.

Contents

1 Introduction 4

2 CASL Overview 8

2.1 CASL Operation 8

2.2 A CASL Module Definition 12

2.2.1 TITLE and LET definitions 12

2.2.2 PLIST structure definitions 14

2.2.3 PLIST instantiations 14

2.2.4 Sequence identification section 14

2.2.5 Initialisation of PLISTs 15

2.2.6 Sequence definitions 15

2.3 Complete CASL example 16

3 CASL: A Critique 24

3.1 CASL as a Process Control Language 24

3.2 CASL as a High Level Language 27

3.2.1 Local Variables 27

3.2.2 Interface variables 28

3.2.3 Looping Constructs 29

3.2.4 Real Numbers 31

3.2.5 Foreign Language Procedure Calls 31
3.2.6 Symbolic Labels 31
3.2.7 Macros and File Inclusions 32
3.2.8 Boolean Expressions 32
3.2.9 Layout characters 33
3.2.10 IF Statement 35
3.2.11 LET Definition 36
3.2.12 String Manipulation 36
3.2.13 Long Strings 36
3.2.14 PLIST initialisation 37
3.2.15 PLIST and SEQUENCE numbers 38
3.3 Conclusion 38

4 Implementation Issues **40**
4.1 Preprocessor and Lexical Scanner 40
4.2 Parser 41
4.3 Code Generation 45
4.4 Tag name to plant address conversion 45
4.5 Testing 46

5 Intermediate Language and Code Generation **47**
5.1 CASL Intermediate Language 47
5.2 Code Generation 50
5.2.1 Code Optimisation 50

6 Conclusion **52**

A CASL REFERENCE SECTION **54**

A.1 Assignment Statement 54

A.2 Plant Control Statements 54

A.3 Conditional Statement 55

A.4 Program Execution Control Statements 55

A.5 Input and Output Statements 56

A.6 Job State and Job Position Statements 57

A.7 Timing Statements 59

A.8 Error Notification and Handling Statements 60

A.9 Miscellaneous Statements 60

B CASL BNF DESCRIPTION **62**

B.1 The BNF for CASL+ 63

C VIRTUAL MACHINE DEFINITION **71**

D CASL RESERVED WORDS **101**

Chapter 1

Introduction

Batch processing provides manufacturers with the flexibility of producing many different products using the same basic equipment, compared to continuous processes which are used whenever large quantities of a product are required and the specification remains relatively constant [24]. The recipes (or sequences) for producing these batch products need to be updated continually in order to maximise the use of raw materials and plant equipment, thus producing cost-effective products. Batch process control has presented unique challenges to the process control fraternity[20], [24], [10] and [27], compared to continuous control which is generally well understood and easier to implement [24].

Many low-level and unsophisticated systems have evolved over the years that allow the computer to automatically control a factory or industrial plant [18] and [10]. These systems were often difficult to understand and to implement and the need arose for a high level language with concepts and constructs that could map onto the typical plant operations. CASL[1] (**C**ontrol **A**nd **S**equencing **L**anguage) [3], is such a special-purpose language and is used for programming the various functions associated with batch process control. CASL forms part of the CYGNUS[2] process management and control system [2] and was developed in the seventies by ICI (Ltd) in England.

CASL allows the programmer to perform the following operations:

- Sequencing of batch processes;

[1] Pronounced "castle" or "cassel"

[2] Registered trade mark of AECI Limited

- Continuous control of plant variables;
- Supervisory control calculations;
- Operator communication;
- Data logging to printouts or batch records.

Figure 1.1 illustrates the interaction of the various components in the CYGNUS system. It shows a simple, single node configuration. However, it can easily be extended to a distributed multinode configuration.

The system includes a translator, loader, run-time interpreter, and operator display and control facilities. A CASL source module is passed through the *translator* which checks for syntax and semantic errors and produces an output file in an intermediate language. This binary output file is submitted to the *loader* which loads the module into the sequence database partition in memory. The loader stores the module's physical memory location in a shared library area. This area is accessed by the sequence *interpreter* at run-time, which processes the instructions and data in the sequence partition. The interpreter interacts with the real-time database (and therefore the plant equipment) by reading and/or setting the attributes of the plant equipment as represented in the database. A *scanner* task is responsible for the communication between the plant equipment and the real-time database. The *Operator Communication Package* (OCP) maintains a display of the status of all the jobs currently executing on the plant. The operator is able to inspect any job and to monitor its progress. A sequence may transmit text output to the operator and receive input in return, which allows the operator to interact with the execution of a job if necessary. Output may also be directed from the sequence interpreter to a file on disk or the system printer for creating reports and batch records.

As will be seen later, CASL is a somewhat archaic language and there are some factors that impair a programmer's ability to write good batch control programs in CASL. The distributor of CYGNUS, AECI Process Computing, approached the Department of Computer Science at the University of Cape Town with a view to developing a local CASL compiler, while at the same time modernizing the language where possible while still maintaining upward compatibility. Thus the objectives of this dissertation are to

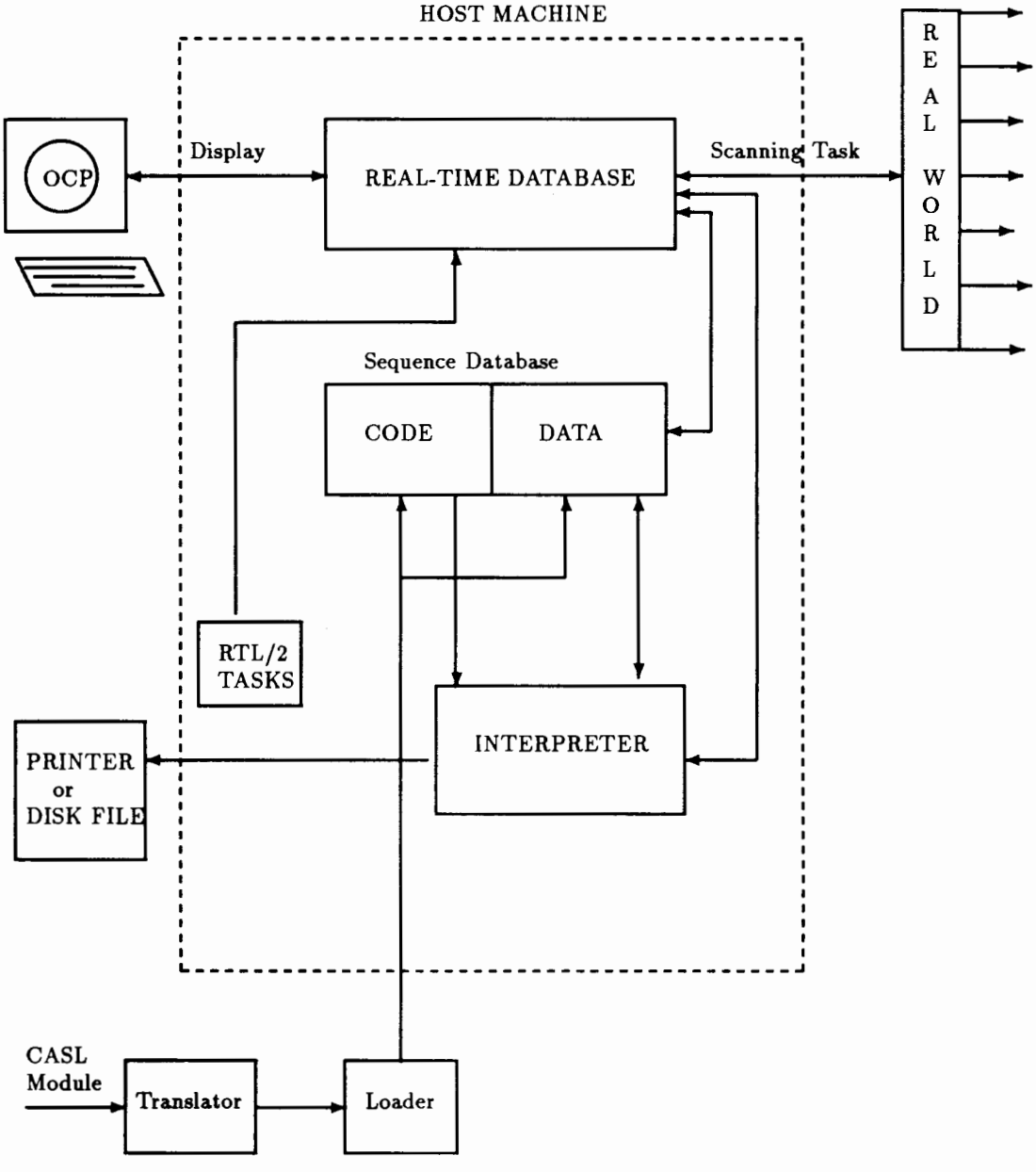

Figure 1.1: CASL Environment

- examine CASL and identify functionality, readability and module clarity improvements that can be made to the language, keeping in mind the investments already made in CASL sequences;
- design a new language, CASL+[3], that is upwardly compatible with CASL; and
- implement a compiler for the new language, using compiler construction tools such as *lex* and *yacc*.

CASL is analysed as a process control language and as a high level language which resulted in enhancements being proposed and implemented to improve the language, bearing in mind the existing investment in CASL. The first chapter presents a brief overview of the language to introduce the unfamiliar reader to CASL. Secondly, a critique of CASL as a specialised high level batch control language is presented. The third chapter deals with the implementation issues that arose during the development of a compiler for CASL+. The final chapter deals with a description of the intermediate language generated by the compiler.

[3]This name is only used for the purpose of this document to distinguish between the "old" and "new" language.

Chapter 2

CASL Overview

This chapter introduces the concepts and major components of CASL. It is not intended as a language tutorial, but rather to provide the reader with an overall understanding of the language.

2.1 CASL Operation

In a process control plant a number of operations (or processes) typically take place at the same time. A batch process is defined by Rubin [27] as "having a recognizable start and finish, with one or more states or phases occurring during the operation." In CASL, these operations are represented by *jobs* (similar to tasks) which are executed (pseudo-) concurrently. A CASL job can be described as the execution of a *sequence*. A *sequence* is similar to a procedure in a conventional high level language and consists of a sequence or block of executable CASL statements. In addition to being CALLed (like a procedure call), a sequence may also be STARTed, in which case a new job is created. A job may consist of a number of sequences because of possibly nested CALLs to other sequences. However, there is only one active sequence per job (see Figure 2.1).

Sequences are re-entrant, therefore more than one job can access the same sequence simultaneously. A sequence is subdivided into *steps* by certain CASL statements. These steps play an important role in the concurrent execution of jobs. While it is the responsibility of the process engineer to subdivide a sequence into steps, in practice these steps occur naturally due to the distribution of certain CASL statements in the sequence code.

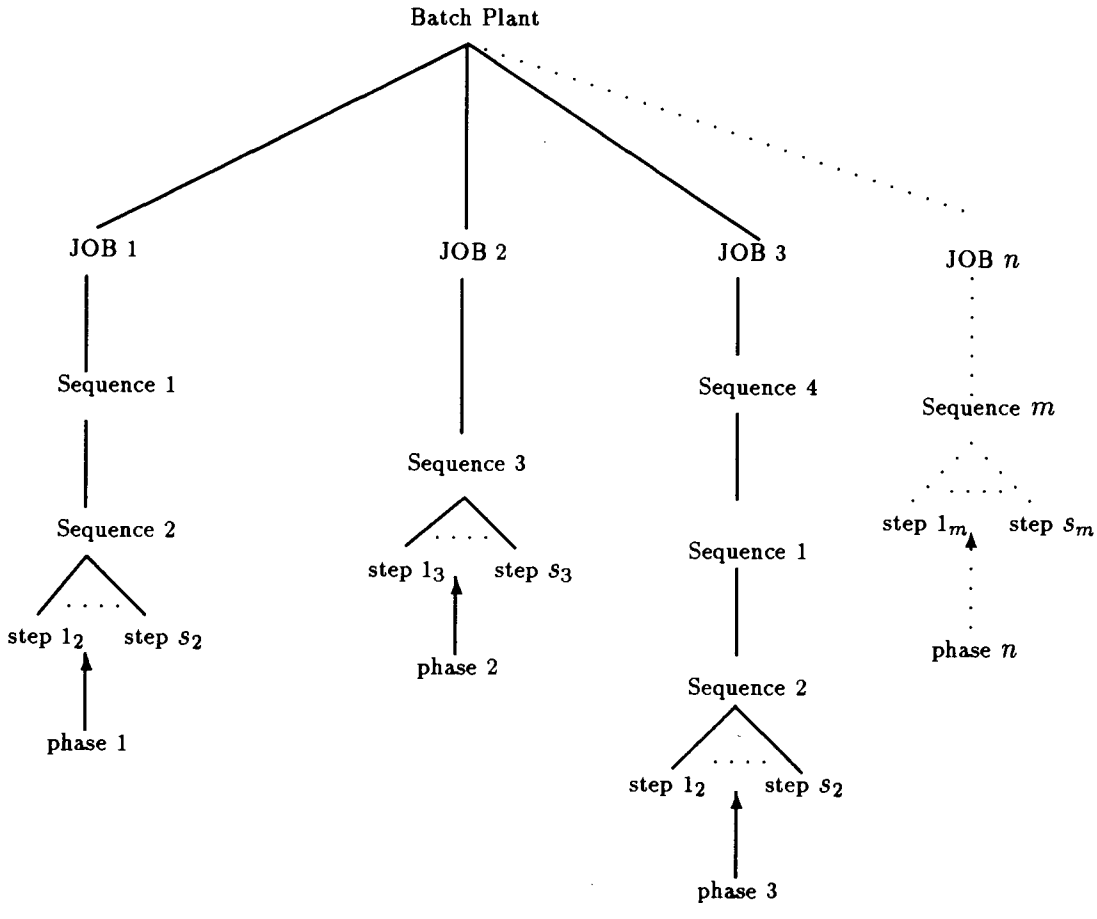

Figure 2.1: Hierarchical plant execution diagram

Pseudo-concurrency is obtained by executing part of every job at least once in a given time interval (e.g. once per second). This concept is called a *heartbeat.* The part of a job that is executed during a single heartbeat is called a *phase* and corresponds to the execution of a single step in a sequence of code. In other words, one phase of every job is executed every heartbeat. For example, Figure 2.1 illustrates the conceptual structure of a CASL system being executed in a batch plant. There are n jobs in the plant (CASL allows a maximum of 60), each with one or more sequences forming part to it. A new job is created when a sequence is STARTed, and this sequence is known as the base sequence. A job may consist of more than one sequence due to nested CALLs to other sequences, e.g. JOB 3 consists of Sequence 4 (the base sequence) which CALLed Sequence 1, which in turn CALLed Sequence 2. Note that Sequence 2 is accessed (being executed) by JOB 1 and JOB 3. In a single heartbeat n phases will be executed, e.g. step i_2 of Sequence 2 as part of JOB 1, step j_3 of Sequence 3 as part of JOB 2, step k_2 of Sequence 2 as part of JOB 3, etc.

Phases are not preempted, but run to completion. Mutual exclusion is therfore guarnanteed within a step. Synchronisation and scheduling of jobs are obtained with special CASL statements. The behaviour of these statements with respect to the current state of a job is illustrated by Figure 2.2.

A new job is created using the CASL START statement, this job is then said to be in the ACTIVE state. The current state of a job may be ACTIVE, FREE, DELAYED or HELD. The DELAYM and DELAYS CASL statement change the state of a job from ACTIVE to DELAYED for the time period specified in minutes or seconds. An active job may be placed on HOLD(with the HOLD statement) until it is explicitly activated by another job, with the CONTINUE statement. Finally a job may be freed by using the CASL KILL statement which terminates the job.

Synchronisation of jobs is achieved by using three CASL statements, namely SETHOLD, FUTHOLD and CONTINUE. For example, job A and job B are dependent upon each other. Job B is unable to proceed beyond label 100 until job A has reached label 150. To ensure synchronisation, job A may issue a FUTHOLD statement to place job B in the HELD state when it reaches label 100. Thus the execution of job B is suspended until job A reaches label 150 and which then issues a CONTINUE statement to activate job B. The SETHOLD statement, may also place any given job in the HELD state. The current job being executed

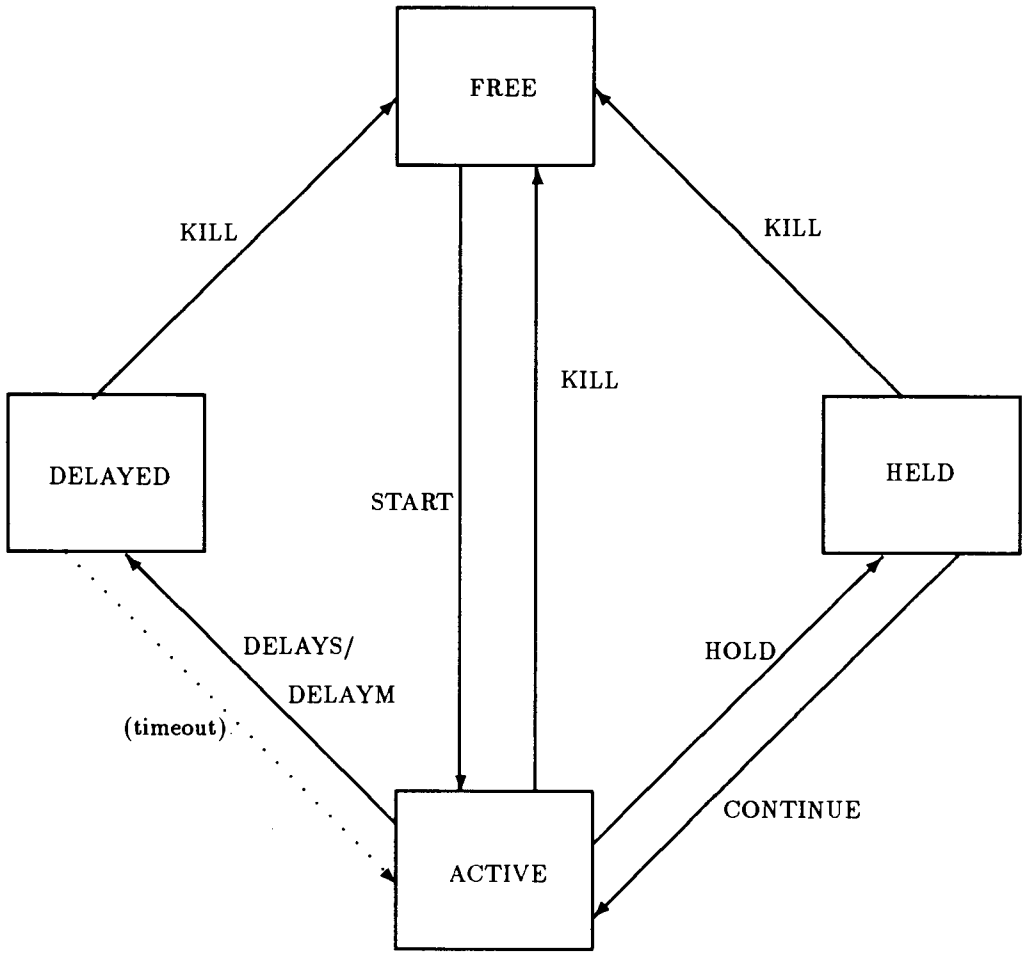

Figure 2.2: Transitions between Job states

may also "know" that it cannot proceeed until some other event has occured and therefore place itself on hold, with the trust that another job will activate it when it is safe to proceed. This is achieved when the current job issues a HOLD statement. The code extract below illustrates the scenario described above:

```
JOB A                          JOB B

   :                              :     % Going into held state
   FUTHOLD(B, 100)             100:     % On hold until released
   :                              :     % Job B activated by Job A
150:
   CONTINUE(B)
   :
```

The data on which sequence logic operates can be separated into three parts: process database data; PLIST data; and job-local data. *Process database data* allows the sequence logic to reference plant equipment, read the plant state, and control the equipment by accessing values (attributes associated with plant addresses) in the database. A PLIST *data block* is a block of data (or parameter list) that has been grouped together and given a name. A PLIST can be passed as a parameter to a sequence, or can be referenced explicitly turning it into a global data block. A limited number (only four) *job-local* variables are provided. They have fixed names (`X1` to `X4`) and are local to a job. A reference to such a variable from within any sequence will reference the corresponding variable in the job executing that sequence.

2.2 A CASL Module Definition

A CASL application project is usually split into a number of modules. Each module consists of the six sections described next. Some of these sections are optional. An example of a module is given in Figure 2.3. It has the following components:

2.2.1 TITLE and LET definitions

The `TITLE` statement defines a module title, while `LET` definitions define text replacement strings (parameterless macros).

```
TITLE "CASL/TST/18FEB90" EXAMPLE MODULE

LET SCREEN = OCT 100;              %OCP dialogue
LET NL = 10;                       %NEWLINE character for printouts

PLISTDEF                           %Plist definition section
PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG
PLIST BATCHREC = INT BATCHNUM, SHIFTNUM, MESS SEQTITLE, SEQID

PLISTNUM                           %Plist declaration(instantiation) section
1   :  REACTOR(1)
5   :  REACTOR(4)
13  :  REACTOR(9)
100:   BATCHREC(1)

SEQNUM                             %Sequence definition section
10:   MAINSEQ
20:   SUBSEQ REACTOR, BATCHREC
30:   ANOTHERSEQ BATCHREC

DATA                               %Plist initialisation section
REACTOR(4) = '2XV326, '2XV315, 50.0, 60.0, 0.0(2), 22
BATCHREC(1) = 0(2), "Transfer of raw materials for process 10", "X21208"

SEQUENCE MAINSEQ
10:
      IDENT1 "Mainseq"
      DEVIN SCREEN
      DEVOUT SCREEN

20:   X1 = ME                      %Set local variable X1 to the current job no.
      START SUBSEQ REACTOR(4), BATCHREC(1)
      GOTO 10
ENDSEQ

SEQUENCE SUBSEQ AREACTOR, ABATCH
      IDENT1 "Subseq"
      :
      ABATCH.BATCHNUM = 100
      OPEN AREACTOR.INLET
      :
ENDSEQ
```

Figure 2.3: A sample CASL module.

2.2.2 PLIST structure definitions

PLIST structure definitions appear in the **PLISTDEF** section and are like "named TYPE" declarations in Pascal. Every PLIST must have a unique name and contains a list of field definitions. Each field must have one of five *modes* (or types), namely **INT** (integer), **NUM** (real), **BITS** (bit flags), **PLAD** (plant address), or **MESS** (message – actually string), and must have a name that is unique within the module. An array is defined by placing the size of the array after the mode keyword. For example

```
PLISTDEF
PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG
PLIST BATCHREC = INT BATCHNUM, SHIFTNUM, MESS SEQTITLE, SEQID
```

defines any **REACTOR** PLIST to consist of two plant addresses (**OUTLET** and **INLET**), two arrays of two real numbers each (**REQVOL** and **AREA**), and an integer (**ERRFLAG**). Similarly **BATCHREC** PLIST consists of two integers (**BATCHNUM** and **SHIFTNUM**), and two string variables (**SEQTITLE** and **SEQID**).

2.2.3 PLIST instantiations

The section headed by the keyword **PLISTNUM** is used to declare the actual PLISTs (PLIST instantiations) for the module. The **PLISTNUM** section of Figure 2.3, for example, defines three PLISTs with the **REACTOR** structure, namely **REACTOR(1)**, **REACTOR(4)**, and **REACTOR(9)**, and one PLIST with the **BATCHREC** structure. Each instantiated PLIST is created with a list of variables that correspond to the fields defined in the PLIST-structure.

In addition to the instantiation number (the numbers in parentheses), each PLIST must be identified with a unique integer identifier. These identifiers are used by the interpreter as indices to reference the PLIST items in the data partition of the sequence database in memory. Each of these identifiers must be unique in a project in order to prevent a conflict from occurring during access. A maximum of 255 PLISTs may be declared in CASL.

2.2.4 Sequence identification section

All sequences used in a module, whether they are defined in the module or just referenced in it, must be numbered and listed in the **SEQNUM** section. Each entry in the section

consists of a number followed by a colon, the sequence name, and a list of PLIST-structures that defines the kind of parameter list(s) associated with the sequence. This is similar to procedure prototypes found in other languages. The integer identifiers are used by the interpreter as indices into the code partition of the sequence database. For example the **SEQNUM** section from the sample module, identifies three sequences, one (**ANOTHERSEQ**) of which is external to the module. **MAINSEQ** does not make use of a PLIST, while **SUBSEQ** requires two PLISTs, namely a **REACTOR** type PLIST and a **BATCHREC** type PLIST.

```
SEQNUM
10:  MAINSEQ
20:  SUBSEQ REACTOR, BATCHREC
30:  ANOTHERSEQ BATCHREC
```

2.2.5 Initialisation of PLISTs

The **DATA** section contains the optional initialisation of variables in instantiated PLISTs declared in the **PLISTNUM** section. The PLIST is identified by its name and instantiation number and is followed by a list of data values. The list of data values are separated by commas must have a one to one correspondence with the PLIST-structure definition in the **PLISTDEF** section.

The list of data values may include a replication factor. This is specified as an integer in parentheses following a data item. For example, in Figure 2.3, the variables in the PLIST **REACTOR(4)** are initialised as follows:

```
REACTOR(4).OUTLET    = '2XV326
REACTOR(4).INLET     = '2XV315
REACTOR(4).REQVOL(1) = 50.0
REACTOR(4).REQVOL(2) = 60.0
REACTOR(4).AREA(1)   = 0.0
REACTOR(4).AREA(2)   = 0.0
REACTOR(4).ERRFLAG   = 22
```

2.2.6 Sequence definitions

A sequence definition starts with the keyword **SEQUENCE** followed by the sequence name and a list of formal parameter names. These formal parameters refer to PLISTs of the kind

defined in the **SEQNUM** section. The sequence definition is terminated with the keyword **ENDSEQ**. CASL labels and executable statements are enclosed between these two keywords. For example:

```
SEQUENCE ASEQUENCE APLISTPARA, ANOTHER
   ⋮
   Block of CASL statements
   ⋮
ENDSEQ
```

CASL executable statements can be sub-divided into a number of functional groups. *Assignment* statements change the contents of PLIST and local variables. Assignments can be made to variables of any type except **MESS**. However, type conversion is only allowed between integers and reals. *Plant control* statements allow the manipulation of **PLAD**'s and their attributes in the database. For example, the value of an attribute may be changed with the **SET** statement. *Program execution control* statements direct the flow of control through a sequence. These include an IF THEN ELSE statement, a simple looping construct, a subroutine (sequence) call, as well as a **GOTO** statement. *Text input and output* statements read and write text data from and to the current input and output devices.

Job control statements can be used to affect the state and execute positions of not only the current job, but also other jobs. Most of these statements have been mentioned under job synchronisation. A further example is the **DIVERT** statement which is similar to a **GOTO**, except that it causes another (specified) job to **GOTO** the given label. *Timing* and *error notification* statements form the last two groups of CASL statements. A complete discussion of all the CASL statements may be found in APPENDIX A.

2.3 Complete CASL example

The following example module forms part of a large real-time application project for a chemical plant. It is included here as an example of a real life CASL module. The comments in the code serve as annotation for some of the statements. The sequence TRANSFER212 listed below performs the following functions:

1. Checks that the **INLET**, **OUTLET** & **DEMIN** water valves are closed, the reactor is drained, and the agitator is available. If not, the operator is informed and the sequence awaits his go-ahead.

2. Prompts the operator for the **DEMIN** quantity to be transferred and then calculates the equivalent (target) level to monitor during the transfer.

3. Opens the **DEMIN** valve until the required volume is transferred. Then closes the **DEMIN** valve.

4. Attempts to start the agitator. After 1 second, checks to see whether the agitator has actually started.

5. Opens the **FLO7** feed valve.

6. Checks that the operator has confirmed transfer completion.

7. Closes the **FLO7** inlet valve and "ends" the sequence.

```
TITLE "X21208/PLA/04AG87" SAREF SEQUENCE : SIDESTREAM "C" TRANSFER
%-------------------------------------------------------------------%
%  Set up some system conventions using the "LET" command           %
%-------------------------------------------------------------------%
LET OPENED = 1;
LET CLOSED = 0;
LET ON     = 1;
LET OFF    = 0;
LET RUN    = 1;
LET STOP   = 0;
LET AVAIL  = 1;
LET TRIPPED= 0;
LET ALARM  = WRITE 3;
LET OCPIO  = DEVIN OCT 100      % OCP dialogue %
             DEVOUT OCT 100;
LET PRINTER= 2;
LET NL     = 10;                % NEWLINE character for printouts %
LET FF     = 12;                % FORMFEED character for printouts %
LET EOS    = OCT 200;           % End-of-stream %
LET ACK    = 6;                 % Request operator acknowledge %
LET SCC    = 123;               % Start colour change sequence %
LET ECC    = 125;               % End colour change sequence %
LET SET_TAG= X1=;
LET SET_JOB= X2=0
             RTL 10;

%-------------------------------------------------------------------%
%  Now define the "PLIST" data structures.  These "PLISTS" (parameter%
%  lists), allow sequences to be re-used by merely changing the data %
%  used, without recoding.                                          %
%-------------------------------------------------------------------%
PLISTDEF
```

```
PLIST REACTOR  = PLAD OUTLET, INLET, DEMININ, LEVEL, AGAVAIL, AGSTAT,
                      AGCMD, STEAM, CONDENSATE, CONDENSBYPASS,
                      YPUMPAVAIL, YPUMPSTAT, YPUMPCMD,
                      FPUMPAVAIL, FPUMPSTAT, FPUMPCMD,
                      TEMP, AIRV, PRESSURE, REDOX,
                      DEMISTISOL, DEMISTWASH,
                 NUM  REQDVOL, CURRLEV, AREA, REQDLEV, MAXTEMP,
                      REDOXLIM, HLEVEL, LLEVEL, CURRREDOX,
                      CURRTEMP, SHIFTDEMIN, MAXPRESS,
                 INT  ERRFLAG, MAXFILLTIME


PLIST FLAGS    = PLAD ALM08, ALM09, ALM10, DLG08, DLG09, DLG10,
                  INT  SEQFLAG, ALMFLAG001,
                       RMODFLAG1, RMODFLAG2, RMODFLAG3, RMODFLAG4,
                       ALMFLAG208, ALMFLAG209, ALMFLAG210

PLIST BATCHREC = INT  AREANUM, SHIFTNUM, BATCHNUM, TIME,
                      TXFSTARTTIME, TXFENDTIME,
                      REDSTARTTIME, REDENDTIME,
                      FILTSTARTTIME, FILTENDTIME,
                 MESS OPNAME, DATE, SEQTITLE, SEQID

PLISTNUM       % Declare the ID numbers of the "PLISTS" :  %

1:  REACTOR(1)
2:  FLAGS(1)
3:  BATCHREC(1)

SEQNUM         % Declare SEQUENCE 8, AREA 212's transfer sequence :  %

8:  TRANSFER212

%-------------------------------------------------------------------%
%  Initialise data to be used in the PLIST                          %
%-------------------------------------------------------------------%
DATA

REACTOR(1)  = '212XV326, '212XV315, '212QV307, '212L321, '212AG06/A,
              '212AG06/S, '212AG06/R, '212XV318, '212XV313, '212XV3
              '212PP07/A, '212PP07/S, '212PP07R,
              '212PP08/A, '212PP08/S, '212PP08R,
              '212T319, '212XV305, '212P306,
              '212A322, '212XV325, '212XV331,
              0.0, 0.0, 20.0, 0.0, 40.0, 100.0, 550.0, 100.0,
              50.0, 12.0, 0.0, 150.0,
              0, 20

FLAGS(1)    = '212ALM08, '212ALM09, '212ALM10, '212DLG08, '212DLG09,
              '212DLG10,
              0, 0, 0, 0, 0,
              0, 0, 0, 0

BATCHREC(1) = 212, 0, 0, 0,
              0, 0,
              0, 0,
              0, 0,
              "RON REAGAN", "10-JUL-86",
```

```
            "SIDE STREAM <C> TRANSFER RECORD",
            "X21208"

%-----------------------------------------------%
% S E Q U E N C E    D E F I N I T I O N      %
%-----------------------------------------------%
SEQUENCE TRANSFER212

%---------------------------------------%
% S T A R T    T H E    S E Q U E N C E %
%---------------------------------------%

1:
    SET_TAG 'BOB-1          % These instructions set the current JOB number %
    SET_JOB                 % into the Type 7 tag specified for this job...  %

    IDENT1 "212TXF"         % These "IDENT" strings are used to %
    IDENT2 "STARTUP"        % prefix messages to the operator.  %
                            % IDENT1 identifies the sequence %
                            % IDENT2 identifies the current phase %
    DEVOUT PRINTER          % Set up the default output device %
    AREANUM = 212

5:
    %-------------------------------------%
    %  Cleanup for next run               %
    %-------------------------------------%

    SET LEVEL.MVE = 0.0
    CURRLEV = 0.0
    SET AGSTAT.OPN = OFF
    SET OUTLET.OPN = OPENED
    SET INLET.OPN = OPENED
    SET DEMININ.CLS = CLOSED

    DELAYS 5                % Initially, the sequence is "HELD" %
    HOLD                    % until the operator kicks it off.  %

    %-----------------------------%
    %  I N I T I A L I S A T I O N  %
    %-----------------------------%

    IDENT2 "INITIALISE"
    ERRFLAG = 0             % Clear the error flag initially %
    TIMEOUT 12 10           % Set up a 10 second timeout, diverts %
                % To label 12 on timeout %

10:
    TOD TXFSTARTTIME        % Note the start time of the sequence %
    BATCHNUM = BATCHNUM + 1
    CLOSECK OUTLET, INLET,  % Close valves & check for 10 sec.%
            DEMININ, AGCMD
                            % If OK, goto following step %
    TIMEOUT                 % Reset timeout mechanism %
    DELAYS 5
    GOTO 20                 % ...If the valve checks all passed %

12:
    %---- Notify the operator which checks failed :  ------%
```

```
    IF OUTLET.CLS ISNT CLOSED
       ALARM "Outlet valve", OUTLET, "not closed"
       ERRFLAG = 1
    END

    IF INLET.CLS ISNT CLOSED
       ALARM "Inlet valve", INLET, "not closed"
       ERRFLAG = 1
    END

    IF DEMININ.CLS ISNT CLOSED
       ALARM "Demin valve", DEMININ, "not closed"
       ERRFLAG = 1
    END

    IF LEVEL.MVE > 0.01*LEVEL.OHE
       ALARM "Reactor not drained"
       ERRFLAG = 1
    END

    IF AGAVAIL.OPN ISNT AVAIL
       ALARM "Agitator", AGAVAIL, "not available"
       ERRFLAG = 1
    END

    IF ERRFLAG # 0
       OCPIO                  % Set up the default output device %
       TEXT "Rectify errors and press #SCC#BY#ECC#CONTINUE#SCC#C#ECC# "
       "when ready#EOS#"
       DEVOUT PRINTER         % Set up the default output device %
       GOTO 5
    END

% V A L V E    C H E C K S    S U C C E S S F U L    S O ... %

%-------------------------------------------------%
% G E T    T H E    D E M I N    Q U A N T I T Y %
%-------------------------------------------------%

20:
    IDENT2 "DEMINQTTY"
    OCPIO                     % Set up the default output device %
    TEXT "Enter DEMIN WATER transfer volume (litres) :  "
    READ REQDLEV
    TEXT "Press #SCC#BY#ECC#CONTINUE#SCC#C#ECC# to proceed with transfer#EOS#"
    DEVOUT PRINTER            % Set up the default output device %
    HOLD

25:
    IF REQDLEV > HLEVEL
       OR REQDLEV < LLEVEL
       OCPIO                  % Set up the default output device %
       ALARM "Invalid transfer volume, new volume ?#EOS#"
       DEVOUT PRINTER         % Set up the default output device %
       GOTO 20
    END

% Having determined the transfer targets, proceed :  %
```

```
%-------------------------------------------------------%
% T R A N S F E R    D E M I N    T O    R E A C T O R %
%-------------------------------------------------------%
30:
    IDENT2 "DEMINTXF"
    OPEN DEMININ
    TIMEOUT 38 MAXFILLTIME

35:
    CURRLEV = CURRLEV + 30.0   % A very simple tank model %
    SET LEVEL.MVE = CURRLEV

36:
    IF LEVEL.MVE >= REQDLEV
       TIMEOUT    % Reset the timeout %
       GOTO 38
    END
    % ELSE ...  %
    GOTO 35                    % ...  and continue filling %

37:
    OCPIO                      % Set up the default output device %
    ALARM "Excessive filling time"
    TEXT "Investigate & press #SCC#BY#ECC#CONTINUE#SCC#C#ECC# when ready#EOS#"
    DEVOUT PRINTER             % Set up the default output device %
    HOLD
    GOTO 35

38:
    CLOSE DEMININ
    WRITE 1 "Water transfer valve closed"
    DELAYS 4

%---------------------------------%
% S T A R T    A G I T A T O R %
%---------------------------------%
40:
    IDENT2 "AGITATION"
    SET AGSTAT.OPN = OFF
    DELAYS 4
    OCPIO
    ALARM "Agitator", AGSTAT, " failed to start"
    TEXT "Press #SCC#BY#ECC#CONTINUE#SCC#C#ECC# for agitator restart#EOS#"
    SET '212TXFSWI.MVE = 5.0
    HOLD
    SET AGSTAT.OPN = ON
    DELAYS 1

44:
    IF AGSTAT.OPN IS STOP
       OCPIO                   % Set up the default output device %
       ALARM "Agitator", AGSTAT, "not running"
       TEXT "Press #SCC#BY#ECC#CONTINUE#SCC#C#ECC# for agitator restart#EOS#"
       DEVOUT PRINTER          % Set up the default output device %
       HOLD
       GOTO 40
    END
```

```
46:
    DELAYS 10
    OPEN INLET
    WRITE 1 "Cake transfer started"
    DELAYS 10

%-------------------------------------------------------------------------%
%   C O N F I R M A T I O N     O F     C A K E     T R A N S F E R       %
%-------------------------------------------------------------------------%
50:
    IDENT2 "CONFCTXF"
    OCPIO                    % Set up the default output device %
    WRITE 2 "If cake transfer is complete, "
    "press #SCC#BY#ECC#CONTINUE#SCC#C#ECC# "
    SET '212TXFSWI.MVE = 4.0
    DEVOUT PRINTER           % Set up the default output device %
    HOLD

55:
    CLOSE INLET

%--------------------------------------%
%   E N D     O F     S E Q U E N C E   %
%--------------------------------------%
60:
    IDENT2 "ENDSEQ" TOD TXFENDTIME
    SHIFTDEMIN = SHIFTDEMIN + REQDLEV
    DEVOUT PRINTER           % Set up the printer output device %

62:
    TEXT "#FF#BATCH REPORT"  % Batch Reporting %
    NEWLINES 1
    TEXT "************"
    NEWLINES 2
    TEXT "-----------------------------------------------------------------#NL#"
    TEXT "AREA NUMBER : "
    PRINT AREANUM 3
    NEWLINES 1
    TEXT "SEQUENCE ID : ", SEQID
    NEWLINES 2
    TEXT "TITLE : ", SEQTITLE
    NEWLINES 2
    TEXT "DATE : ", DATE
    SPACES 28
    TOD TIME
    TEXT "TIME : "
    PRINT TIME
    NEWLINES 1
    TEXT "-----------------------------------------------------------"
    NEWLINES 2
    TEXT "OPERATOR : ", OPNAME
    NEWLINES 1
    TEXT "SHIFT NO. :  #EOS#"
    PRINT SHIFTNUM
    NEWLINES 1
    TEXT "BATCH NO. :  "
    PRINT BATCHNUM
    NEWLINES 1
    TEXT "START TIME : "
```

```
    PRINT TXFSTARTTIME
    NEWLINES 1
    TEXT "END TIME : "
    PRINT TXFENDTIME
    NEWLINES 1
    TEXT "TRANSFER VOL : "
    PRINT REQDLEV
    TEXT " (LITRES) "
    TEXT "#NL#SHIFT CUMUL VOL : "
    PRINT SHIFTDEMIN
    TEXT " (LITRES) #NL,FF,EOS#"

64:
    GOTO 5                    % The sequence goes back to 5 to a "HOLD" state %
    ENDSEQ
```

Chapter 3

CASL: A Critique

CASL can be evaluated as a batch process control language and as a high level language. In the former, one would ask how well does the language serve the process personnel in specifying control sequences that must be performed in the plant. In other words, how good is the mapping from CASL statements to plant operations? In the second evaluation one would consider such aspects as the clarity and simplicity of the language concepts, clarity of module syntax (i.e. readability), naturalness for the application, ease of module creation and use [26].

3.1 CASL as a Process Control Language

In an attempt to evaluate a batch control language a set of criteria need to be established, against which the language may be compared. CASL is thus evaluated against a set of features that should be incorporated into a good batch control language.

As mentioned before, batch process control is difficult. Reasons for this difficulty can be found in [20] and [24]. In [12] an interpretive language for sequence control of chemical processes is discussed and it is once again highlighted that a batch control system must be able to handle a wide product variability due to changing market needs. It is for this reason that [10] rightly states that "it is not sufficient to use a general purpose high level language for batch control", like, for example Pascal or RTL/2. Thus the specialised high-level language is becoming a vital tool in the computer sequence control system.

Shaw [29] states that by using his generalised program structure application programmers can create well engineered batch programs. This program structure includes five basic sections, "the initiation section, the control section, the operator override logic, the error fault handling section, and the termination section." A high level batch process control language should be able to provide the programmer the ability to incorporate these sections into a control program. CASL allows the programmer to structure all application programs into the five sections defined by Shaw. In addition the following features should be present in a good batch process control language.

- High level commands. These are specialised commands that only find their application in the realm of process control. e.g. open or close a piece of equipment.
- On-line sequence changes should be possible to allow minimal plant disruption during recipe changes. Not only would it be preferable to perform these On-line but they must be able to be done easily and with the least amount of effort.
- In-line documentation. Experience has shown that little of the design philosophy of a software project is documented. Thus it is imperative that code be self documenting, easily readable and understandable to the process personnel rather than the computer specialists. This feature is not achievable with the low level languages.
- Re-entrancy. The language should allow defined blocks of sequence code to be re-entrant. Thus the same sequence code may be executed simultaneously by more than one job (or task).
- Need for operator intervention. This is necessary when a critical decision needs to be made about an unknown error condition, or when the operator needs to provide input to the program about the nature of a job, either its initiation, control or termination.
- Parameterisation of sequence code. This together with the feature of re-entrancy allows different instantiations of the same parameter list to be passed to the same sequence code. Consider the following scenario: a sequence is defined to manufacture a batch of paint. There are four different paints and five unique colours of each paint to be made, the sequence of operations is the same to manufacture each type and

colour paint. In this situation it should be possible to define the types and colours of the paints in the parameter lists which are passed to a single block of sequence code to manufacture the paint.

The following discussion will show that CASL may be considered as a good batch process control language. CASL provides the programmer with a number of high level commands that find special application only in the realm of process control. The ability of the programmer to open or close instrumentation on the plant, repeating the operation until it succeeds is an example of such a high level command found in CASL. A description of all CASL statements may be found in Appendix A.

On-line sequence changes may be performed easily in CASL causing minimal plant disruption. The sequence and data definitions are declared separately, allowing these sections to be loaded into separate memory segments. A loader forms part of the CYGNUS package which provides the process engineer the ability to replace an existing sequence definition with the modified sequence definition. This provides for easy recipe changes to be made while the rest of the plant continues to function normally. Similarly the process engineer may also replace the PLIST data block on which a sequence depends.

Comments may be placed freely in the code of a CASL program. In addition many of the statement names describe their operation, which adds to the readability of the code. However, CASL restricts the use of the newline character in the layout of statements to certain predefined positions, which does affect the ability of the programmer to provide in-line documentation.

CASL allows many jobs to be executed simultaneously. The mechanism was described earlier; how a portion of each job is executed in a given time slice. This creates the effect that all the jobs are being serviced in parallel. Since CASL sequences are also re-entrant, the same logic may be simultaneously executed by a number of jobs.

The OCP (Operator Communication Package) allows the programmer to communicate with the operator, by displaying useful information and receiving input from the operator, using his console. CASL in turn has the ability to specify the source and destination device of its input and output. For example, directing audit trail information to the system printer, critical and non-critical information to the OCP. In critical situations the

program may allow the operator to intervene by causing control to be dependent on his response to the situation.

As mentioned a PLIST data block is a block of data (or parameter list) that has been grouped together and given a name. Since CASL allows a number of instantiations of the same PLISTs to be defined, a sequence may be invoked with the same or a different PLIST instantiation. In the scenario described in the list of features of a batch process control language under parameterisation of sequence code, the colours and types of a paint may be defined in separate PLISTs which are then passed to the same sequence to manufacture the paint.

CASL has provided support for those features that should be included as part of a good batch process control language. Other well known sequence control languages as listed by [12] are for example Kent's K-70 and K-90, PROSEL, Foxboro's FOX-2 and BATCH. An attempt was made to acquire in depth information about these and similar languages, however this was unsuccessful since the local distributors of these products did not want to disclose such information for investigation and comparison purposes.

3.2 CASL as a High Level Language

Whereas CASL does well as a process control language, it has a number of deficiencies when judged as a high level language. In the following sections these shortcomings are examined and solutions proposed for each of them.

When changing a well-used language, one always has to keep existing users of the language in mind. Some of the unfortunate features of CASL cannot be addressed without making the new language incompatible with the old. Other problems can be solved in such a way that the new language is upwardly compatible with the old.

Those issues that can be addressed while maintaining upward compatibility are discussed first and then those to which the solutions have more serious implications are pointed out.

3.2.1 Local Variables

CASL allows only four local (fixed-named, and typeless) variables per job. The main reason for this restriction was the severe memory restrictions the original system had to

cater for. This is no longer the case and CASL+ has been enhanced with proper local variables.

Proper local variables can be introduced through a special kind of PLIST, namely a LOCAL PLIST. LOCALs are then declared at the start of a sequence, before the executable statements. The same format is followed as that of a PLIST declaration in the PLISTDEF section of the module. For example,

```
SEQUENCE MAINSEQ
LOCAL PLAD(16) LOCPLAD,
      INT      LOCINT1, LOCINT2, LOCINT3,
      MESS     MESSLOC
   :
   X1 = LOCINT1 + LOCINT2 * LOCINT3
   :
ENDSEQ
```

The same usage and naming convention applies as for normal PLISTs, except for the scope of the variables. While the names of local variables must be unique, their scope are limited to only the sequence in which they are defined. Therefore, a local variable in sequence A is not visible to sequence B in the same job, and neither is a local variable in sequence A of job J visible to sequence A in job K. This is achieved by ensuring that every time a sequence is invoked, an instantiation of the local PLIST is created and destroyed when the sequence exits. This is not unlike invocation records and stack frames in conventional programming languages.

3.2.2 Interface variables

CASL allows only PLISTs to be passed as parameters to sequences. It is not difficult to extend this to allow any variable to be passed as a parameter. CASL+ refers to these additional parameters as *interface variables*. They may be used in combination with PLISTs. For example,

```
PLISTDEF PLIST SOMEPLIST = ...
PLISTNUM 1:  SOMEPLIST(1)
SEQNUM
10:  MAINSEQ
```

```
20:  SUBSEQ(INT, PLAD) SOMEPLIST

SEQUENCE MAINSEQ
LOCAL INT COUNTER
   :
   CALL SUBSEQ(10, COUNTER, 'HV-100) SOMEPLIST(1)
   :
ENDSEQ

SEQUENCE SUBSEQ(INT IVINT1, COUNT, PLAD VALVE) APLIST
   SET VALVE.MVE = 100
   APLIST.SOMEFIELD = ...
   :
ENDSEQ
```

Here IVINT1, COUNT and VALUE are formal interface variables used together with a PLIST APLIST in SUBSEQ sequence. 10, COUNTER and 'HV-100 are the actual interface variables past to sequence SUBSEQ. Interface variables can be implemented as yet another type of PLIST – at each CALL or START an instantiation is made of an interface PLIST containing copies of the actual interface variables. Interface variables are passed by value and as with local variables, the interface PLIST is destroyed when the sequence terminates.

3.2.3 Looping Constructs

CASL does not have any flexible looping construct apart from the GOTO statement and the REPEAT ... FINISH loop, which is restricted to a maximum of 255 iterations. One of the reasons for this is the possibility that a loop can take long to complete (e.g. a large repeat count) and if it does not contain an end of phase statement, the job may overrun its time slice in the heartbeat. (A backwards GOTO forces an end of phase, therefore loops constructed from GOTOs do not have this problem.) In spite of this, more powerful looping constructs should be provided. Suitable programming practices such as putting conditional phase terminators inside loops can prevent overruns. As a compromise between simplicity and versatility, CASL+ supports a FOR loop and a WHILE loop. For example,

- FOR Loop.

  Syntax:

```
FOR <counter> = <initial_value> TO <final_value> STEP <step_value> DO
⋮
NEXT <counter>
```

Example:

```
FOR I = 1 TO 20 STEP 2 DO
   X1 = I + 10
   JOBSTAT(X1).TANKPRESS = 0.0
   FOR J = 5 TO 7 DO
      TEXT "Debug statement, inside the FOR J loop"
      OPENCK PLADLIST
      WRITE 6 "Pladlist has been opened"
   NEXT J
NEXT I
```

- WHILE Loop.

Syntax:

```
WHILE <condition> DO
⋮
ENDWHILE
```

Example:

```
I = 1
WHILE I <= 20 AND ERROR > 0 DO
   J = 5
   WHILE J > 1 DO
      IF OUTLET.MVE >= 200 THEN
         WRITE 9 "The pressure is too high"
         WRITE 9 "in the tank", OUTLET
         ERROR = 0
      END
      J = J - 1
   ENDWHILE
   I = I + 1
   WRITE 2 "Debug statement, end of while i <= 20"
ENDWHILE
```

The semantics are similar to those in Pascal or Modular-2. The FOR loop counter must be a positive PLIST item or local variable. Similarly the STEP value is required to be positive and the value one is used as a default. Both loops may be nested.

3.2.4 Real Numbers

CASL supports a 16-bit NUM which stores a floating point number in a (non-standard) internal format. Values can range from 0.00003 to 65504.0 with an accuracy of one part in 2048. In order to obtain better accuracy and to make more efficient use of floating point co-processors where available, 32-bit IEEE format floating point numbers are supported by CASL+. The current design of the intermediate language requires that all basic data types have the same size (i.e previously two bytes). This is due to the fact that a data item is accessed via its position in the data list and not via its offset. Thus all basic data types have been expanded to 32-bit values.

3.2.5 Foreign Language Procedure Calls

CASL allows RTL/2 procedures to be called from within a sequence with the statement

```
RTL n
```

where n is an integer identifying the procedure to call. Parameters and results can only be passed through the 4 local variables. There is no reason why these procedures cannot be identified symbolically. Support should also be provided for other foreign languages such as C. CASL+ defines foreign language procedure call statements as follows,

```
XCALL C procedure_name
XCALL RTL procedure_name
```

3.2.6 Symbolic Labels

CASL allows only numeric labels in the range of 0-255 since the original implementation required any label to be identifiable by a single byte. Since memory restrictions are no longer a big issue, the new language should allow the use of symbolic (alpha-numeric) names for labels. This enhancement to the language will increase the readability and the clarity of CASL+ modules. For example.

```
SEQUENCE MAINSEQ

SEQ_SETUP:              %Replaces 10:
   ⋮
ALARM:                  %Replaces 250:
```

```
   WRITE 9 "ALARM: Sequence will abort automatically"
   KILL ME
   GOTO SHUTDOWN

FATAL_ERR:                  %Replaces 251:
   WRITE 9 "OPERATOR PLEASE TAKE CONTROL OF PLANT SHUT DOWN"
   RETURN

SHUTDOWN:                   %Replaces 255:
   UNCHECK
   RETURN
ENDSEQ
```

3.2.7 Macros and File Inclusions

A separate preprocessor has been developed for CASL+ that will execute all preprocessor commands. CASL only supported a simple text replacement facility and a file inclusion command. The text replacement facility has been expanded to allow multiple levels of nested LET definitions. However, the major improvement has been the introduction of parameterised macro expansion facility. This allows the programmer to pass arguments to the LET definition. e.g.

```
LET IO(X) = DEVIN X
            DEVOUT X;
```

then IO(OCT 200) is expanded in the CASL source, to

```
DEVIN OCT 200
DEVOUT OCT 200
```

The file **inclusion** command has been expanded to allow nested files to be included into the source file. **Hooks** have been provided for a number of other preprocessor commands to be implemented in the future, these include conditional compilation, undefining a current LET definition and position stamps.

3.2.8 Boolean Expressions

A complex condition may become complicated if there are a number of **AND/OR** operators. The readability and clarity of such a condition is improved by the introduction of parenthesis. CASL does not support parentheses to group conditions. However, CASL+ allows

the inclusion of parentheses in a complex condition which will aid the understanding of the IF statement. For example,

```
A OR B AND C OR D
```

could be interpreted either as

```
(A OR B) AND (C OR D)
```

or as

```
A OR (B AND C) OR D
```

CASL evaluates the expression as indicated in the second example, by ANDing B and C, with the result ORed to A and finally ORed to D.

All the issues discussed so far can be (and have been) addressed by implementing the proposed changes while still maintaining upward compatibility with the old language. Solutions to the issues in the following sections, however, will have more serious implications.

3.2.9 Layout characters

CASL does not use an explicit statement terminator or separator like the semicolon in C or Pascal. While CASL statements are always terminated by a newline character, not all newline characters terminate statements. One of the reasons given for not using explicit statement terminators [14] is that most users of CASL don't know programming languages and therefore statement terminators would be "unnatural" for them and an unnecessary complication. However, with the CASL syntax the way it is, the use of newline characters actually complicates matters: a set of rules is required specifying where newlines must, may, and may not be used. For example, newlines are allowed after any comma and before (but not after) ANDs and ORs in conditional expressions. It would have been much simpler to state newlines may be used anywhere where spaces can be used. It is somewhat embarrassing to have to explain to someone that she may write

```
IF A.SIZE >= B.SIZE
   AND A.SIZE >= 0 THEN
```

but not

```
IF A.SIZE >= B.SIZE AND
   A.SIZE >= 0 THEN
```

Had it not been for the syntax of two statements, namely the CALL and START statements, it might have been possible to allow free-format CASL programs without explicit statement terminators (while still using an LALR(1) parser). The problem lies in the fact that the two fragments

```
CALL CLEAN_UP TANK(1) = 0
```

and

```
CALL CLEAN_UP
TANK(1) = 0
```

are both legal (non-free-format) CASL fragments, but mean completely different things. In the former **TANK** is a PLIST with a single field that is initialised to 0 in the **CALL**. In the latter, **TANK** is an array variable (a member of a PLIST – if there is only one instantiation of the PLIST, the field name is enough to identify the variable) that is assigned the value 0.

Even without free-format the (poor) syntax of CALL and START statements gives problems to an LALR(1) parser. As indicated before, PLISTs may be passed as parameters in these statements, and furthermore, they may be (optionally) initialised at that time. One could therefore have the following

```
PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG
PLIST BATCHREC = INT BATCHNUM, SHIFTNUM, MESS SEQTITLE, SEQID
⋮
CALL SUBSEQ REACTOR(1) = '2XV326, '2XV315, 50.0, 60.0,
                         0.0(2), 22, BATCHREC(1)
```

which initialises the fields of **REACTOR(1)** to the same values of **REACTOR(2)** in the sample module of Figure 2.3. To further complicate matters, the initial values may be references to other PLIST fields, e.g.

```
PLIST BOB = INT(20) A
⋮
START X1 BOBSEQ BOB(1),BOB(2)=1,,3(5),BOB(4).A(2),(7), 10(5), BOB(4)
BOB(3).A = 10            % a statement following the call
```

The work of not only the parser, but also the human reader would be much easier if the initialisation of a PLIST is enclosed in braces ({}) and repetition factors are enclosed in square brackets ([]). The above example could then be written as follows:

```
START X1 BOBSEQ BOB(1),
                BOB(2) = {1, , 3[5], BOB(4).A(2), [7], 10[5] },
                BOB(4)
BOB(3).A = 10            % a statement following the call
```

3.2.10 IF Statement

The IF statement is defined by a condition, a body and the **END** keyword. However, the consistency and clarity of an IF statement is clouded by the programmers option to include the keyword **THEN**, between the condition and the body of the statement. In other words the IF statement has the structure

IF <*condition*> <*optional_then*> <*statements*> **END**

To a reader of the source code it is not obvious where the condition ends and the body of the IF statement begins. The keyword **THEN** should be enforced in the syntax of the IF statement. The omission of the **THEN** keyword coupled with the fact that the equality and the assignment operator are the same, can be misleading. For example if the programmer has an equality condition in the IF statement and omits the **THEN** keyword, with the first statement in the body being an assignment statement, the syntax obscures the semantics, as in

```
IF X1 = 1            % if X1 is equal to 1
   X4 = X1           % THEN assign X1 to X4
   PRINT X1          % etc.
   ⋮
END
```

The **THEN** keyword in a future syntax definition should be mandatory.

3.2.11 LET Definition

Unlike other CASL statements, **LET** statements are terminated with an explicit terminator – a semi-colon. While this allows for definitions such as

```
LET CLEAR_FLAGS =
   X4 = 0
   REPEAT 70
      X4 = X4 + 1
      FLAG(X4) = 0
   FINISH;
```

readability can be improved if an explicit continuation character is used and no termination character (to fit in with the rest of the language), e.g.

```
LET CLEAR_FLAGS =   \
   X4 = 0           \
   REPEAT 70        \
      X4 = X4 + 1   \
      FLAG(X4) = 0  \
   FINISH
```

3.2.12 String Manipulation

CASL does not support string assignments directly. In order to assign a string to a (message) variable, one has to use a special version of the **WRITE** statement, e.g.

```
WRITE 255 MESSVAR, "New text string"
```

(Every **WRITE** statement has a priority which determines the urgency of the WRITE statement. The priority range is from 0-9. The WRITE statement used with the special priority of 255 indicates a string assignment.) It would be much simpler if one could write

```
MESSVAR = "New text string"
```

3.2.13 Long Strings

A string constant is defined to be a set of string characters enclosed in a pair of double quotes. A string character is any character except double quotes ("), a hash symbol (#)

and a newline character. A long string may be written on successive lines but each line must be enclosed in double quotes. LET replacements may be made within a string if the LET symbol is enclosed within a pair of hash symbols (# #). The current syntax definition of a long string may appear to the programmer as a number of unique strings on separate lines instead of a single string. However, if a hash character replaced the closing quote and the opening quote of successive strings the list of successive strings may more easily be recognised as a single string. The following example illustrates the problem and a possible solution, which is also supported by Barnes on page 23 of [7].

```
WRITE 6 "This is a very long string that is used to illustrate the"
"fact that long strings can get very long."
"The syntax should be changed to highlight to the programmer "
"that this is in fact one long string."
```

should be changed to

```
WRITE 6 "This is a very long string that is used to illustrate the #
#fact that long strings can get very long.  #
#The syntax should be changed to highlight to the programmer #
#that this is in fact one long string."
```

3.2.14 PLIST initialisation

The header section of a CASL module consists of a number of items, of which **DATA** and **PLISTNUM** are only two. The **PLISTNUM** section declares a number of PLISTs of the same type as defined in the **PLISTDEF** section, by associating an instantiation number to the PLIST. The DATA section is then responsible for optionally initialising some of these PLISTs declared. This is an unnecessary fragmentation of information and should rather be combined, for example

```
PLISTNUM
1   :  REACTOR(1)
5   :  REACTOR(4) = '212XV326, '212XV315, 50.0, 60.0, 0.0(2), 22
13  :  REACTOR(9)
100:  BATCHREC(1) = 0(2),
                    "Transfer of raw materials for process 10",
                    "X21208"
```

3.2.15 PLIST and SEQUENCE numbers

As mentioned already, each instantiation of a PLIST must be assigned a unique integer identifier which is used at run-time to reference the PLIST items. Similarly a SEQUENCE must be assigned a unique integer identifier by the programmer in order to reference the sequence at run-time. I propose the introduction of a project file that controls the execution of the project. This file would free the programmer from the need to know and define SEQUENCE and PLIST numbers and serve to centralise the definitions and declaration of data. Currently every module is required to include a PLISTDEF and PLISTNUM section to define and declare those PLISTs that are used in the sequences contained in the module. The SEQNUM section is required to associate the sequence with a number and provide a prototype against which to check usage of that sequence.

The project file would contain all PLIST definitions and declarations for the entire project as well as all the SEQUENCE prototype definitions. It would then be the compiler's responsibility to assign appropriate numbers. Individual modules would be compiled against the project file and the system would ensure that dependent modules are recompiled as appropriate.

3.3 Conclusion

In the preceeding sections it has been argued that CASL is a good batch control language. Re-entrancy of sequence code and the ability to pass PLISTs as parameters to a sequence maps well onto the batch processing environment. A particular sequence (recipe) may be defined and by simply passing a different PLIST (set of ingredients), a different batch of a products may be produced. The abstraction of concurrency concepts namely a heartbeat and phase, allows the CASL programmer to concentrate on the content of a module rather than on concurrency. This together with the instantiation of multiple PLIST definitions, interface with the Operator Communication Package, and the standard interface to the plant database all contribute to the success of CASL as a process control language – evidenced by more than 60 installations of CASL in South Africa. However, as a high level language CASL has a number of deficiencies. Figure 3.1 summarizes the improvements identified and implemented.

Problem	Proposed solution	Implemented
• Four fixed-named local variables	General local variables via LOCAL PLIST	√
• Only PLISTs as parameter	Interface variables	√
• Limited looping construct	**FOR** and **WHILE** statements	√
• Accuracy of real numbers	32-bit IEEE format floating point numbers	√
• Only numbered RTL procedure calls	Support for symbolic calls to C procedures	√
• Only 256 numeric labels	2^{32} symbolic labels	√
• Limited **LET** definitions	Parameters; nested invocation and nested file includes	√
• Boolean operator precedence	Introduction of parenthesis	√
• Lack of free format	An explicit statement terminator	×
• **START** and **CALL** statement syntax	New syntax	√
• Optional **THEN** in **IF** statement	Mandatory **THEN**	×
• Multiline **LET** with terminator	Explicit continuation character	×
• No string assignment	Explicit string assignment	×
• PLIST instantiation and initialisation	Unify the two sections	×

Figure 3.1: Summary of Improvements

Chapter 4

Implementation Issues

A new CASL compiler has been developed that implements all the improvements described before that could be made while maintaining upward compatibility. In this chapter the implementation of the compiler using PCLEX and PCYACC [1] is discussed.

4.1 Preprocessor and Lexical Scanner

PCLEX was used to generate a lexical scanner from regular expressions defining the CASL tokens. An attempt was made to incorporate file inclusions, macro definitions and calls directly into the scanner and parser. This could be achieved by changing the source of the scanner input at appropriate times. However, because PCLEX generates a scanner that uses the more efficient, so-called *flex* algorithm, this was not possible. Instead of reading the input a character at a time, a *flex* scanner reads input a line at a time and then examines the characters in the internal buffer. Redirecting input source under these circumstances became just too complicated, and a separate, hand-coded preprocessor was developed using a public domain C preprocessor as a base.

A minor problem arose during the implementation of the lexical scanner. In the original CASL, two or more strings separated only by a newline are considered as a single string. A string token was therefore defined accordingly. However, this caused the scanner's internal buffer to overflow on long strings. While the buffer size could have been extended, one would not have been able to guarantee that it is big enough since there is no limit on the lengths of strings (in some applications strings of more than 1K bytes are defined). The

problem was solved by moving the recognition of long strings to the parser. Rules were defined to recognise lists of strings only separated by newline characters and to concatenate these strings and store them as a single string.

4.2 Parser

The automatic parser generation tool PCYACC was used to implement the parser for CASL. The reason for using this tool was because of a requirement that the implementation of the compiler should be easily maintainable. For future development and enhancements to the language, an automatic compiler generation tool was selected to aid the programmers. A grammar description file served as input for PCYACC producing a C file which is compiled and linked to the supporting modules to build the compiler.

The definition of the language syntax that was available was incomplete and incorrect. CASL was defined using an extended version of BNF notation. The most notable problem was the inclusion of the semantics in the syntax definition of the language. A complete and accurate definition of CASL may be found in Appendix B.

For example the CASL **CALL** statement illustrates an instance of an incorrect definition. The definition of the **CALL** statement omitted a comma between PLIST parameters of the statement. As illustrated in the example below the statement may be interpreted either as a **CALL** with two PLIST parameters or as **CALL** followed by an assignment statement.

```
CALL SEQ10 VESSEL(1) = 0, 1 TANK(1) = 40

CALL SEQ10 VESSEL(1) = 0, 1
TANK(1) = 40
```

In addition to the language definition being incorrect it was also found to be incomplete. Consider the CASL **KILL** statement, which was defined to have the following syntax:

```
KILL jobno
```

where **jobno** is any integer value. Yet the most commonly used instance of the statement is where the job number is the current job number, which is stored in the key variable **ME**, thus it is clear that the **jobno** should include **keyvar**, which is defined as being one of **ME, ERROR, PRIV** or **TAG**. Thus the following syntax should be legal,

```
KILL keyvar where keyvar is ME, ERROR, PRIV and TAG.
```

As mentioned it was required that the new language be free format. The current language specification forced the use of a newline character as a statement terminator and as a layout character, thus true free format could never be achieved. To ensure free format in the source code the newline character was considered as white space. As a result a few statements were ambiguous and caused conflicts to arise during the parsing process of a CASL source module. A few options were pursued in an attempt to solve the conflicts that arose during parsing due to a lack of statement terminators and ambiguous language definitions. The first solution was thought to be the introduction of a semicolon as a statement terminator. Although this addition to the language would have resulted in incompatibility with the existing compiler, a utility could have been written to convert existing modules to the new language definition. This was unacceptable to the initiators of the project felt that a semicolon was a nuisance and an irritation when programming.

The second option involved the construction of additional grammar rules in the parser, effectively to implement further lookahead in those statements whose syntax was ambiguous due to the lack of a statement terminator. For example, the syntax for the original CASL **PRINT** statement is given below. The syntax definition is presented using an extended BNF form, where non-terminal symbols are enclosed in angle brackets (<>), terminal symbols appear in upper case letters and optional symbols are enclosed in square brackets ([]).

```
<printst>           := PRINT <printitem> [<opt_format_length>]
                    ;
<opt_format_length> := INT
                    ;
```

Furthermore a label is defined as

```
<labelst> := INT ':'
          ;
```

This leads to a *shift-reduce* conflict illustrated by the following two **PRINT** statements:

```
PRINT intvar 10        % Print invar, format width of 10

PRINT numvar           % Print numvar format width of 12
10:                    % Numeric label 10
```

The first statement prints the value of an integer using the specified format width of ten characters. The second statement uses the default format width of twelve characters to print the real number, the following statement is the numeric label ten. An LALR(1) parser cannot decide whether or not the ten is the start of the label statement or if the ten is the format width of the **PRINT** statement. This problem can be solved by adding lookahead rules:

```
<printst>           := <print_item> [<opt_format_length>]
                    ;
<opt_format_length> := INT [<opt_label>]
                    { if (flag) then default else label}
                    ;
<opt_label>         := ':'
                    {flag == TRUE}
                    ;
```

Here a flag is set in the semantic actions when the : of a label is seen and this information is then used by the earlier rule to distinguish between a label and a format length specifier. Similar rules can be used to solve ambiguity found in other statements.

However, there were two statements that could not be solved by simply adding lookahead grammar rules to the grammar. Semantic information is essential in order to determine the meaning of the **CALL** and **START** syntax. Consider the syntax of the CASL **CALL** statement as given in the original language definition.

```
<callst>          := CALL ID [<opt_plist_para>]
                  ;
<opt_plist_para>  := ID '(' INT ')' [<opt_values>]
                  | [<opt_plist_para>] ',' ID '(' INT ')' [<opt_values>]
                  ;
<opt_values>      := '=' <expression_list>
                  ;
```

These rules can be illustrated by the example we saw in section 3.2.9 (layout characters).

```
CALL CLEAN_UP TANK(1) = 0

CALL CLEAN_UP
TANK(1) = 0
```

The first consists of a single PLIST initialisation in the CALL statement for the sequence CLEAN_UP. While the second example is a CALL statement with no PLIST parameters followed by an array element assignment statement. In this example it is necessary to know that TANK is a PLIST in order to associate it with the sequence, since it could also be an array element initialisation. Thus no extra syntax rules could possibly interpret this example. As another example, the following CASL CALL statement,

```
CALL SEQ VESSEL(1) = 0,1,"hello", GENERAL(1), PIPE(3)
```

could be interpreted as

```
CALL SEQ {VESSEL(1) = 0,1,"hello",GENERAL(1),PIPE(3)}
```

or as

```
CALL SEQ {VESSEL(1) = 0,1,"hello"}, {GENERAL(1)}, {PIPE(3)}
```

where in the first case the statement consists of a single initialised PLIST parameter with five data items, while in the second case, the statement has three PLIST parameters, with only the first being initialised.

From these examples it is clear that it is impossible to syntactically analyse all CASL statements correctly while accepting the newline character as white space. Therefore, it was decided to retain the use of the newline character as a statement terminator and layout character. This enabled the parser to correctly interpret all CASL examples except for an instance of the CALL and START statements. Even an explicit statement terminator did not allow these statements to be correctly analysed (see section 3.2.9). The only solution to the problem was to change the syntax of the language, by introducing layout characters to allow unambiguous interpretation of the CALL and START statements. The previous example would have the following appearance, using the new syntax of the language.

```
CALL SEQ VESSEL(1) = {0,1,"hello"}, GENERAL(1), PIPE(3)
```

Although this change to the language affected upward compatibility with the existing language, it was felt that it could be justified since in practice, the initialisation of PLIST parameters occurred very seldomly in CALL and START statements. If the old statement syntax is used the compiler will produce an error message instructing the programmer to use the new syntax.

4.3 Code Generation

The CASL source language is translated into an intermediate language, which is executed by an interpreter. The definition of the intermediate language as received from the initiators of the project was incorrect and incomplete. This and other problems that were solved are dealt with comprehensively in Chapter 5.

4.4 Tag name to plant address conversion

The convention for referencing plant addresses in CASL is by tag name, which is usually application dependent. The standard CYGNUS syntax requires that a tag name begin with a prime (') followed by up to nine significant characters, drawn from the letters A–Z, digits 0–9 and @. The layout characters - and / may be used freely, e.g. 'FIC-410/A, '@MASSFLOW, and '77-HV-101/A. A tag name that is used in a CASL module must be present in the CYGNUS database in order to be valid. Since each PLIST item is represented internally by a 32 bit value (16 bits in the original CASL), tag names, which are the "values" of PLAD variables, need to be converted to 32 bit values. In other words, internally, a PLAD is a 32-bit representation of a tag name.

In the old version of the CYGNUS system the conversion from tag name to PLAD, was performed by the translator. This was achieved by simply passing the tag name to a special dictionary task, which looked up the name in a dictionary and returned an associated 16 bit number that served as a kind of index in the database. This number was then stored in the PLAD item and used for subsequent database access. The disadvantage of this approach is that if the database is changed, all modules referencing the database have to be re-translated.

In the new compiler, the tag name to PLAD conversion is left to the loader, to be performed at load time. The compiler records every occurrence of a tag name and passes this information to the loader. Multiple occurrences of the same tag name will result in a chain of pointers being formed from the last occurrence to the first. The loader converts the tag name and inserts the packed value into the intermediate code. The loader thus becomes responsible for the validation of tag names.

Run time conversion and validation is not required since there is a restriction that prevents the database from being updated while a sequence that uses it is running.

4.5 Testing

Testing of the compiler was difficult and it cannot be stated that it was completely tested, since no compiled CASL modules have actually been executed. Since no interpreter existed for the new intermediate code that the compiler generated. The most important test of the compiler remained the correctness of the intermediate code generated.

A set of benchmark examples were used to ensure that the code generated by the new compiler matched that generated by the current compiler. This was hand checked to ensure that the same sequence of code was generated for the same set of source code. Where the code generated differed it was ensured that the code performed the same functions and would result in the same behaviour. The code generated for the new CASL statements was checked for logic errors to ensure the desired effect may be achieved when the code is executed.

Chapter 5

Intermediate Language and Code Generation

5.1 CASL Intermediate Language

One of the requirements for the CYGNUS process control package is the facility for easy on-line modification of the control operations. "With this requirement in mind, it was decided to represent the source modules on the on-line computer by an intermediate language." [22] Thus the translator performs an off-line translation of the source module into an intermediate language. A complete description of the intermediate language is given in Appendix C.

There are a number of additional advantages to be gained from the intermediate representation of the source language: firstly retargeting is facilitated and secondly, a machine independent code optimiser can be applied to the intermediate code. Ullman[5] mentions that the intermediate language should have two important properties . The first property is that it should be easy to produce, i.e. the translation from source code to intermediate code should be easy. Secondly, it should be easy to translate into the target program, i.e. easy to interpret and execute.

Although a re-design of the CASL intermediate language was out of the scope of this dissertation, it has been necessary to deal with the language extensively. Ullman [5] mentions two criteria that should be used in the design of an intermediate language. Firstly

the instruction set (operator set) must be powerful enough to implement the operations of the source language. Secondly, smaller instruction sets are easier to implement in the target machine. However, a small instruction set may result in longer sequences of intermediate code. Thus a tradeoff results between the size of intermediate code blocks and the ease of implementation.

One of the major design criteria of CASL was that it should be memory efficient. This had a direct influence on the design of the intermediate language. The instructions (operators) are all represented as bytes and were arranged in to groups for easy decoding in the interpreter. The groups consisted of instructions (opcodes) with either zero, one or two arguments (operands). The 256 limitation here meant that there were not enough opcodes available and that these groups became inconsistent (i.e. similar types of instructions were no longer grouped together). Thus, "Later modifications involved a lot of patching of the interpreter" [11].
The instruction set may be broken into the following groups.

```
OCT 0   - OCT 17  Opcode (operand1, operand2)
OCT 20  - OCT 77  Opcode
OCT 100 - OCT 117 Opcode operand1
OCT 120 - OCT 177 Opcode
OCT 200 - OCT 220 Opcode operand1
OCT 230 - OCT 277 Opcode
OCT 300 - OCT 317 Opcode operand1
OCT 320 - OCT 377 Opcode
```

The opcodes can be divided into the following functional groups. It should be noted that certain instructions have been added to the intermediate language as the need arose resulting in overlapping functional groups.

```
OCT 0   - OCT 57  Miscellaneous (including flow control, subroutine call
                  and start, array manipulation and register
                  initialisation opcodes)
OCT 60  - OCT 71  Job control opcodes
OCT 72  - OCT 107 Job state and attribute opcodes
OCT 110 - OCT 147 Input/Output opcodes
OCT 150 - OCT 177 Not used
OCT 200 - OCT 220 Miscellaneous opcodes (including Process database
                  access opcodes)
OCT 230 - OCT 277 Arithmetic, Logical and Register manipulation opcodes
OCT 300 - OCT 357 Variable access opcodes
OCT 360 - OCT 377 Not used
```

The complete definition of the intermediate language in Appendix C was constructed from only a partial description of the intermediate language as provided by the initiators of the project. In addition no definition of the virtual machine existed for the implementation of an interpreter. It appears that parts of the intermediate language were designed after the implementation of an interpreter. In many instances it is clear that instructions have been added to the language as they have been needed through the years.

One of the features of the intermediate language is that it is easy to produce. The source language maps very closely onto the intermediate language. Consider, for example, the following extract of CASL source code and its corresponding intermediate code:

CASL Source Code	Intermediate Code
`DEVIN OCT 200`	`SetBbyte(177)` `Devin`

`DIVERT jobno labelno`	`SetAbyte(labelno)`
	`SetBbyte(jobno)`
	`Divert`

The second feature that is important when examining an intermediate language is it's ease of execution. The implementation of the CASL intermediate language is often complex due to the fact that it consists of a large instruction set. For example, the execution of the CASL **PAUSE** statement simply results in the generation of the **Pause** opcode. However, the execution by the interpreter of the **Pause** opcode results in significant processing since the opcode signals the end of a phase for the current job.

CASL Source Code	Intermediate Code	Functions
PAUSE	**Pause**	**Save current environment** **Get next active job** **Set up environment for the new job.**

Thus CASL's intermediate language is very different from a typical machine code instruction set with respect to the size of the instruction set and the level of complexity for the execution of the intermediate language.

5.2 Code Generation

The absence of a verified definition of the virtual machine and of the intermediate language necessitated means of deriving the correct translation of the source language. Benchmark examples were used in this verification process by comparing the code generated by the current translator and the new translator ensuring their equivalence.

5.2.1 Code Optimisation

Minimal expression optimisation (also known as evaluation optimisation) [5] is applied whenever complex expressions need to be evaluated in CASL. Consider the following complex expression:

```
(A and B) or C
```

The condition is true if A and B are true or if only C is true. If A is false then there is no need to evaluate the remainder of the sub-expression, thus immediately expression C is evaluated. On the other hand if expressions A and B are true there is no need to evaluate expression C. Thus the minimal path of evaluation is followed through the complex expression in order to improve run-time efficiency.

Secondly, optimisation of the registers A, B, C and BASE are performed. Register optimization is not possible across phases and labels. Since labels allow control to be transferred from another point in the program by the use of GOTO statements. This meant that all optimization information had to be cleared since the flow of control through the code could not be guaranteed. Register and expression optimization are the only types of optimisation possible since there is a very strong mapping between the source and the intermediate language.

An example CASL module is defined to illustrate the intermediate code generated by the compiler from the corresponding source code. The optimised intermediate code is also given to show the reduction in code size.

CASL Source Code	Intermediate Code	Optimised Code
PLISTDEF		
PLIST VENT = INT MAX, VOL		
PLIST TANK = PLAD INLET		
PLISTNUM		
10: VENT(1)		
20: TANK(1)		
SEQNUM		
100: ANYSEQ		
SEQUENCE ANYSEQ		
VENT(1).MAX = VENT(1).MAX+3	Setbase(10)	Setbase(10)
	Fetch(1)	Fetch(1)
	BtoA	BtoA
	SetBconst(3)	SetBconst(3)
	AddInt	AddInt
	SetBase(10)	
	Store(1)	Store(1)
VENT(1).VOL = 30	SetBconst(30)	SetAconst(30)
	BtoA	
	SetBase(10)	
	Store(2)	Store(2)
SET INLET.MVE = VENT(1).VOL	SetBase(10)	
	Fetch(2)	Fetch(2)
	BtoA	BtoA
	SetBase(20)	SetBase(20)
	Fetch(1)	Fetch(1)
	SetAtt(14)	SetAtt(14)
OPEN INLET	SetBase(20)	
	Fetch(1)	Fetch(1)
	BtoPlad	BtoPlad
	Open	Open
WRITE 3 "Tank is open"	StartWrite(3)	StartWrite(3)
	SetMess(10)	SetMess(10)
	WriteMess	WriteMess
	EndWrite	EndWrite
ENDSEQ		

A sample of CASL modules showed that after optimisation the intermediate code size was on average reduced by between 20 and 30 percent. No additional optimisation has been achieved from the previous compiler to the new compiler.

Chapter 6

Conclusion

Batch processs control languages require greater functionality and flexibility than continuous process control languages, because batch process control is characterised by multi-product manufacturing lines which produce relatively small quantities, and which involve frequent product changes. This dissertation has shown that CASL, a specialised high level language for batch control, is a good *batch control* language. The main reasons for this being:

- The abstraction of concurrency concepts;
- Re-entrancy of sequence code;
- Instantiation of multiple PLIST definitions; and the
- Interface with the OCP and the standard interface to the plant database.

The dissertation further evaluated CASL as a *high level* language and found a number of shortcomings. Specific areas for improvements to functionality, clarity and readability have been identified and improvements suggested. The following improvements have been implemented while keeping the new language upwardly compatible with the old:

- The ability to define local variables;
- The ability to pass ordinary variables as parameters;
- More flexible and powerful looping constructs;

- 32-bit IEEE format floating point numbers;
- Foreign language procedure calls;
- Symbolic labels and step names, and
- Parameterised macros and nested file inclusions

The following improvements were not implemented either because of the compatibility issue, or because the initiators of the project saw no immediate need for it.

- Introduction of explicit layout characters;
- Parenthesis to force precedence of operators in boolean expressions;
- Improvement of string manipulation and representation;
- Plist instantiation and initialisation sections combined; and
- The freeing the programmer from numbering PLISTs and sequences.

An LALR(1) compiler that incorporates all the upwardly compatible improvements has been implemented using the compiler construction tools PCLEX and PCYACC. Except for the **START** and **CALL** statements when PLIST parameters are initialised, the new compiler will compile all existing CASL code in order to protect the large investments made in CASL. These investments will now bear greater returns for the investor due to the improvements made to the language.

Appendix A

CASL REFERENCE SECTION

This appendix is intended to supplement Chapter 2, which provided an overview of the original CASL. In that chapter CASL's operation is discussed, as well as the structure and content of a CASL module. A complete discussion of each CASL statement is presented in this appendix. The statement syntax is given in Appendix B.

A.1 Assignment Statement

Variable = expression

In CASL the contents of a PLIST item is changed by means of an assignment statement. The item on the left hand side of the equals sign is assigned the value of the expression on the right hand side. The type of the expression must be the same-as that of the variable on the left hand side. Type conversion is only allowed between integer and real numbers. Note that a item of type **MESS** may not be assigned.

A.2 Plant Control Statements

Plant control statements operate on the plant equipment referred to by **PLADs**. The following are such statements:

SET plad.attribute = expression

The SET statement allows the value of an expression of the appropriate type to be assigned to a PLAD attribute.

OPEN pladlist,

CLOSE pladlist

The statement OPEN (CLOSE) followed by a PLAD or a list of PLAD values sends a series of signals to the plant actuators to set on (or off) the digital output associated with each PLAD.

OPENCK pladlist,

CLOSECK pladlist

The OPENCK opens all the given PLADs (same procedure as the OPEN statement), and then checks the attribute OPN of the PLADs to ensure all the PLADs have been opened. If all the PLADs have been opened, the next statement is executed, otherwise the current phase is ended and the OPENCK is repeated at the start of the next phase. The CLOSECK is similar, but for closing and the CLS attribute.

A.3 Conditional Statement

CASL only supports an IF conditional statement. A number of conditions may be joined by the AND and OR operators to form a complex condition. The AND has a higher precedence than the OR operator. Two different types of conditions may be evaluated, namely:

- Arithmetic conditions of type INT or NUM which consist of two expressions of the same type operated on by a standard operator.
- Conditions of type BITS. A BITS condition must consist of two BITS expressions operated on by the operators IS and ISNT or the keywords ALLON and ALLOFF which are "bitlists" comprising of all ones and all zeroes respectively which replace the right hand side expression.

A.4 Program Execution Control Statements

Integer :

A label may be specified by an integer in the range 1 to 255 followed by a colon.

GOTO labelno

A Goto statement causes the interpreter to jump to the label number specified by the Goto.

REPEAT repeatcount NEWLINE statements FINISH

The statement encloses a block of statements which are executed a specified number of times. The number of iterations may not exceed 255. The statement may not be nested within another REPEAT statement and branching to labels inside the enclosed block will be undefined as the value of the repeat counter will not be known.

CALL sequenceid plistlist

The CALL statement is the CASL subroutine call. It causes execution to be passed to another sequence. The calling sequence may pass PLIST's (which may be initialised) as parameters to the called sequence. When the called sequence is finished, execution continues after the CALL statement.

RETURN,

ENDSEQ

These statements cause execution of the current sequence to be terminated and all timeouts that are set are cancelled. Then control is returned to the calling sequence. If the sequence executing one of these instructions is the base sequence, execution of the job is terminated.

A.5 Input and Output Statements

DEVIN integer,

DEVOUT integer

These statements are used to set up the current input and current output devices to be used for text input and output.

TEXT stringlist

The statement is used to output character strings.

NEWLINES integer,

SPACES integer

They improve the layout of the output by causing the specified number of newline and space characters to be generated.

VDU x-cord y-cord

The **VDU** statement allows for formatting on VDU screens. However, it is not implemented in the current system.

PRINT printitem format_width

The **PRINT** statement is used to output on the current output device the value of variables, which can be of type **NUM**, **INT**, **PLAD** or **BITS**. The default format length is 12 characters for the **PRINT** statement. This may be changed by providing an integer parameter to the **PRINT** statement.

IDENT1 string,

IDENT2 string

These statements allow the programmer to set up two strings in the sequence that will identify the current activity of the sequence. They form part of the output of various output generated by the statements.

WRITE write_status item_list

The **WRITE** statement produces a message on the system printer. This message consists of any number of literal string and variables of any type. An integer parameter is passed to the **WRITE** statement which indicates the level of urgency of the message.

READKEY variable

The statement awaits the pressing of one key on the input keyboard and returns the key's value in the named integer variable.

READ variable

The **READ** statement attempts to read a value from the current input device. The value will be interpreted according to the type of the specified variable.

A.6 Job State and Job Position Statements

START jobno sequenceid plistlist

The statement is used to activate a job and associate a sequence with it (this sequence is called the base sequence). PLISTs (which may be initialised) can be passed to the sequence as parameters. The job that is activated is placed in the ACTIVE state.

HOLD,

SETHOLD jobno

The statements cause a job's state to be changed to held and a message to be generated to the system printer. SETHOLD causes the specified job to be held, HOLD causes the current job to be held,

FUTHOLD jobno labelno seqid

The FUTHOLD statement causes the specified job to be held once it reaches a specified label.

CONTINUE jobno

The statement changes the state of a job from HELD to ACTIVE.

SSHOT jobno

The statement causes the specified job to go through one phase of the current sequence and to place the job in the HELD state at the end of the phase.

KILL jobno

The KILL statement terminates the execution of the specified job.

STEPON jobno labelno

If the specified job is in the HELD or DELAYED state, STEPON repositions the job at the specified label and changes the state to ACTIVE.

DIVERT jobno labelno

The DIVERT repositions the specified job at the label name in the base sequence and leaves the job in the ACTIVE state irrespective of the previous state.

SETTAG jobno integer

The interpreter associates an integer value TAG with each job in use. When one job starts another job by the START command its TAG value is also allocated to the new job. Sequences can reset TAG values by use of the SETTAG statement which sets the TAG for a specified job to a new integer value.

SETPRIV jobno integer

Each job has a privilege associated with it a privilege. Privilege levels can be reset

in CASL by means of the SETPRIV statement, in which the level is defined by an integer constant or variable.

PAUSE

The PAUSE statement is used to terminate a phase. It is used where other statements do not cause an end of phase, to ensure that the job does not have too long a time slice.

A.7 Timing Statements

TIMEOUT timeoutaction integer

The TIMEOUT statements are used to enclose a block of statements and to monitor the time taken to execute them. There are four different ways to use the TIMEOUT statement, namely:

- The TIMEOUT followed by HOLD and an integer. If another timeout is not encountered in the specified number of seconds the interpreter puts the job into the HELD state.
- The TIMEOUT followed by CONT and an integer. If the timeout elapses, a standard message is output but no other action is take
- The TIMEOUT followed by a step number and an integer. If the timeout elapses the job's position is changed to the specified step number.
- The TIMEOUT statement alone cancels all timeouts in operation.

DELAYS integer,

DELAYM integer

These statements are followed by an integer which specifies in minutes or seconds the duration for which the job is placed in the DELAYED state. After the delay has elapsed the job's state reverts to ACTIVE.

MTIME integer,

MELTIME integer

These statements are used to measure the elapsed time across parts of a sequence. The pair of statements MTIME and MELTIME are used to measure the time in minutes while similar statements STIME and SELTIME measure the time in seconds.

TOD integer

The TOD statement stores the time of day in the named integer variable.

A.8 Error Notification and Handling Statements

It is normal CASL usage to perform explicit checks while doing manipulation of outputs to the plant. Assuming no errors after the output to the plant the sequence does not want to waste time doing continual checking on the plant actuators. Instead it assigns the appropriate actuator to a list in the CHECK command. This enables "event checking" for tags in the list, allowing a centralised CYGNUS facility to do the necessary synchronous checks much more efficiently than the sequence logic.

CHECK checkaction integer plads

It is often necessary to initiate some action when a measurement infringes its high or low limit or when a discrepancy occurs between a digital PLAD set and its actual position. These checks are performed by an event checking scheme when initiated using the CHECK statement. The statement specifies the action to be taken when one of the checks fail. The failure of a check also causes an alarm flag to be set which alerts the operator.

UNCHECK jobno plads

The UNCHECK statement removes checks from the list for the specified job.

A.9 Miscellaneous Statements

GETSEQ sequence_array integerlist variablelist

The GETSEQ statement is used to obtain a number of items of useful information about a job and store this information in a named integer array.

SETSEQ integerlist variablelist

SETSEQ is an alternative to the START statement, using the sequence and parameter list numbers instead of names. The statement starts a sequence using data specified by two lists of integers.

RTL integer

A **RTL** statement transfers control from the interpreter, which is obeying the sequence statements, to a numbered RTL/2 procedure in a list supplied by the user. Parameters can be transferred in local working variables **X1** to **X4**.

EVENT integer

An **EVENT** statement sets the K^{th} marker in an event list accessible by the operating system. The sequence then proceeds to the next statement.

IFNOT condition integer THEN statements END

The **IFNOT** statement is a combination of the **IF** and the **TIMEOUT** statements. The condition is evaluated; if it is true the statement after the keyword **END** is entered. If it is false, the current phase is ended, and a timeout is set up. The evaluation is then repeated at the beginning of successive phases until one of the following events occur:

- The condition is found to be true within the timeout limit. Execution then passes to the first statement following the conditional block.
- The timeout elapses. In this case the conditional block of statements is entered and a timeout alarm message is initiated.

Appendix B

CASL BNF DESCRIPTION

This is a complete and accurate description of the syntax of CASL. A number of symbols have been used to help clarify the syntax definition. The left-hand side of a rule is separated from the right-hand side by an arrow. All non-terminal symbols are enclosed by a pair of angle brackets, whereas terminal symbols are printed in bold face. For example,

⟨preamble ⟩ → **TITLE**

The rule defines a non-terminal symbol **preamble** on the left handside, which produces a single terminal symbol **TITLE**. Square brackets indicate that the symbol or group of symbols are optional. Curly braces followed by an asterix indicate that the symbol or group of symbols are optional and may be repeated any number of times. Curly braces followed by a plus sign indicate that the symbol must be specified at least once and potentially many times. For example,

⟨**sequence** ⟩ → ⟨header ⟩ **NL** [⟨localst ⟩] { ⟨statement ⟩}∗ **ENDSEQ** [**NL**]

⟨constant ⟩ → [**INT** | **NUM**]
|| **PLAD**
|| {**MESS**}+

The first rule defines the structure of a sequence where the sequence header definition and newline token are followed by an optional local statement. The non-terminal ⟨**statement**⟩ may be omitted or repeated many times. The second rule consists of several productions, where in the last production the token **MESS** can be repeated, but must appear at least

once. The double pipe character indicates a new production for the non-terminal on the left-hand side. A single pipe character separates a list of symbols grouped together, e.g. **INT** and **NUM** compose a group.

B.1 The BNF for CASL+

⟨module ⟩ → ⟨preamble ⟩
{ ⟨preamble ⟩}* ⟨plist def section ⟩
{ ⟨preamble ⟩}* ⟨plist number section ⟩
{ ⟨preamble ⟩}* [⟨sequence number section ⟩]
{ ⟨preamble ⟩}* [⟨data section ⟩]
{{ ⟨preamble ⟩}* ⟨sequence ⟩}*

⟨preamble ⟩ → **TITLE**

⟨plist def section ⟩ → **PLISTDEF NL** { ⟨plist def list ⟩}*

⟨plist def list ⟩ → **PLIST ID =** ⟨definition list ⟩ **NL**

⟨definition list ⟩ → ⟨simple mode ⟩ [**(INT)**] **ID**
|| ⟨definition list ⟩ , **ID**
|| ⟨definition list ⟩ , ⟨simple mode ⟩ [**(INT)**] **ID**

⟨simple mode ⟩ → **BITS TYPE**
|| **INT TYPE**
|| **NUM TYPE**
|| **MESS TYPE**
|| **PLAD TYPE**

⟨plist number section ⟩ → **PLISTNUM NL** { ⟨plistdec list ⟩}*

⟨plistdec list ⟩ → **INT : ID (INT) NL**

⟨sequence number section ⟩ → **SEQNUM NL** { ⟨seqnumlist ⟩}*

⟨seqnumlist ⟩ → **INT : ID** [(⟨interface var ⟩)] [⟨plist list ⟩] **NL**

⟨formal list ⟩ → ⟨mode ⟩
|| ⟨formal list ⟩ , ⟨mode ⟩

⟨plist list ⟩ → **ID**
|| ⟨plist list ⟩ , **ID**

⟨data section ⟩ → **DATA NL** { ⟨plist data list ⟩}*

⟨plist data list ⟩ → ⟨plist ⟩ = ⟨data list ⟩ **NL**

⟨plist ⟩ → **ID (INT)**

⟨data list ⟩ → ⟨data value ⟩
|| ⟨data list ⟩ , ⟨data value ⟩

⟨data value ⟩ → [⟨constant ⟩] [(**INT**)]
|| [⟨constant ⟩] [**INT**]

⟨constant ⟩ → **INT**
|| **NUM**
|| **PLAD**
|| {**MESS**}+

⟨sequence ⟩ → ⟨header ⟩ **NL** [⟨localst ⟩] { ⟨statement ⟩}*
ENDSEQ [**NL**]

⟨header ⟩ → **SEQUENCE ID** [(⟨actual list ⟩)] [⟨parameter list ⟩]

⟨actual list ⟩ → ⟨simple mode ⟩ [**(INT)**] **ID**
|| ⟨actual list ⟩ , **ID**
|| ⟨actual list ⟩ , ⟨simple mode ⟩ [**(INT)**] **ID**

⟨parameter list ⟩ → **ID**
|| ⟨parameter list ⟩ , **ID**

⟨statement ⟩ → ⟨stat ⟩ **NL**

⟨stat ⟩ → ⟨assignst ⟩ || ⟨callst ⟩ || ⟨chkst ⟩ || ⟨const ⟩
|| ⟨delayst ⟩ || ⟨devst ⟩ || ⟨divst ⟩ || ⟨eltimest ⟩
|| ⟨eventst ⟩ || ⟨forst ⟩ || ⟨futholdst ⟩ || ⟨getst ⟩
|| ⟨gotost ⟩ || **HOLD** || ⟨identst ⟩ || ⟨ifst ⟩
|| ⟨ifnotst ⟩ || ⟨killst ⟩ || ⟨labelst ⟩ || ⟨nlst ⟩
|| ⟨opst ⟩ || ⟨opchkst ⟩ || **PAUSE** || ⟨printst ⟩
|| ⟨setprivst ⟩ || ⟨readkeyst ⟩ || ⟨readpladst ⟩
|| ⟨readst ⟩ || ⟨repeatst ⟩ || **RETURN** || ⟨rtlst ⟩
|| ⟨setholdst ⟩ || ⟨setst ⟩ || ⟨setseqst ⟩ || ⟨spst ⟩
|| ⟨ssst ⟩ || ⟨startst ⟩ || ⟨stepst ⟩ || ⟨settagst ⟩
|| ⟨textst ⟩ || ⟨timeoutst ⟩ || ⟨todst ⟩ || ⟨unchkst ⟩
|| ⟨vdust ⟩ || ⟨whilest ⟩ || ⟨writest ⟩ || ⟨xcallst ⟩

⟨assignst ⟩ → ⟨assvar ⟩ = ⟨rhs ⟩

⟨rhs ⟩ → ⟨exp ⟩
|| ⟨bitS exp ⟩

⟨assvar ⟩ → ⟨variable ⟩
|| ⟨local var ⟩

⟨variable ⟩ → ID ⟨item ⟩

⟨item ⟩ → [⟨period id selector ⟩]
|| ⟨subscript ⟩ [⟨period id selector ⟩]
|| ⟨subscript ⟩ ⟨subscript ⟩

⟨period id selector ⟩ → . ID [⟨descriptor ⟩]
|| ⟨selector ⟩

⟨descriptor ⟩ → [⟨selector ⟩]
|| ⟨subscript ⟩ [⟨selector ⟩]
|| ⟨subscript ⟩ ⟨subscript ⟩

⟨subscript ⟩ → (⟨subscript value ⟩)

⟨subscript value ⟩ → INT || ⟨variable ⟩
|| ⟨local var ⟩ || ⟨key variable ⟩

⟨key variable ⟩ → ERROR
|| ME
|| PRIV
|| TAG

⟨local var ⟩ → X1 || X2 || X3 || X4

⟨selector ⟩ → . ⟨attribute ⟩

⟨attribute ⟩ → OLE || OHE || ILE || IHE || ET
|| SMT || AL || AUTO || MVE || INSE
|| MT || VAL || INM || INS || GAIN
|| IC || DC || CLS || OPN
|| STAT [⟨subscript ⟩] || OUTP [⟨selector ⟩]
|| INMP [⟨selector ⟩] || INSP [⟨selector ⟩]

⟨exp ⟩ → ⟨term ⟩
|| ⟨exp ⟩ + ⟨term ⟩
|| ⟨exp ⟩ - ⟨term ⟩

⟨term ⟩ → ⟨prim ⟩
|| - ⟨prim ⟩
|| ⟨term ⟩ * ⟨prim ⟩
|| ⟨term ⟩ / ⟨prim ⟩

⟨bitS exp ⟩ → ⟨bit exp ⟩ , ⟨bit exp ⟩
    || ⟨bitS exp ⟩ , ⟨bit exp ⟩

⟨prim ⟩ → [**NUM TYPE** | **INT TYPE**] ⟨primary ⟩

⟨primary ⟩ → ⟨bit exp ⟩
    || ⟨local var ⟩
    || **NUM**
    || ⟨mess constant ⟩
    || ⟨key variable ⟩

⟨bit exp ⟩ → **INT**
    || ⟨variable ⟩
    || **PLAD** [⟨selector ⟩]

⟨callst ⟩ → **CALL ID** [⟨interface var ⟩] [⟨plist list ⟩]

⟨interface var ⟩ → (⟨data exp list ⟩)

⟨plist list ⟩ → **ID** (**INT**) [⟨plist vals ⟩]
    || ⟨plist list ⟩ , **ID** (**INT**) [⟨plist vals ⟩]

⟨plist vals ⟩ → = { ⟨data exp list ⟩ }

⟨data exp list ⟩ → ⟨value ⟩
    || ⟨data exp list ⟩ , ⟨value ⟩

⟨value ⟩ → [⟨exp ⟩] [[**INT**]]

⟨chkst ⟩ → **CHECK** ⟨chk action ⟩ ⟨intval ⟩ ⟨plad list ⟩

⟨chk action ⟩ → **CONT**
    || **DIVERT**

⟨intval ⟩ → **INT**
    || ⟨intval read ⟩

⟨intval read ⟩ → ⟨intval write ⟩ || ⟨key variable ⟩

⟨intval write ⟩ → ⟨variable ⟩ || ⟨local var ⟩

⟨const ⟩ → **CONTINUE** ⟨intval ⟩

⟨delayst ⟩ → ⟨delay key ⟩ ⟨intval ⟩

⟨delay key ⟩ → **DELAYS** || **DELAYM**

⟨devst ⟩ → ⟨device key ⟩ ⟨intval ⟩

⟨device key ⟩ → **DEVIN** || **DEVOUT**

⟨divst ⟩ → **DIVERT** ⟨intval ⟩ ⟨intval ⟩

⟨eltimest ⟩ → ⟨timekey ⟩ ⟨intval write ⟩

⟨timekey ⟩ → **STIME**
|| **SELTIME**
|| **MTIME**
|| **MELTIME**

⟨eventst ⟩ → **EVENT** ⟨intval ⟩

⟨forst ⟩ → **FOR** ⟨variable ⟩ **=** ⟨intval ⟩ **TO** ⟨intval ⟩
[⟨step ⟩] **DO NL**
{ ⟨statement ⟩ }*
NEXT ⟨variable ⟩

⟨step ⟩ → **STEP** ⟨intval ⟩

⟨futholdst ⟩ → **FUTHOLD** ⟨intval ⟩ ⟨intval ⟩ ⟨intval ⟩

⟨getst ⟩ → **GETSEQ** ⟨variable ⟩ ⟨intlist ⟩ ⟨intvallist ⟩

⟨gotost ⟩ → **GOTO** ⟨intval ⟩

⟨identst ⟩ → ⟨identkey ⟩ ⟨mess ⟩

⟨identkey ⟩ → **IDENT1** || **IDENT2**

⟨mess ⟩ → ⟨variable ⟩
|| ⟨mess constant ⟩

⟨ifst ⟩ → **IF** ⟨complex cond ⟩ [⟨then ⟩] **NL**
{ ⟨statement ⟩ }*
{ ⟨elseif ⟩ }*
[⟨else ⟩]
END

⟨then ⟩ → **THEN**

⟨elseif ⟩ → **ELSEIF** ⟨complex cond ⟩ [⟨then ⟩] **NL** ⟨statement ⟩

⟨else ⟩ → **ELSE NL** ⟨statement ⟩

⟨complex cond ⟩ → ⟨cond ⟩
|| ⟨complex cond ⟩ [**NL**] **OR** ⟨cond ⟩
|| ⟨complex cond ⟩ [**NL**] **AND** ⟨cond ⟩

⟨cond ⟩ → ⟨exp ⟩ ⟨comparator ⟩ ⟨exp ⟩
|| ⟨rhs ⟩ ⟨bit op ⟩ ⟨rhs ⟩
|| ⟨rhs ⟩ [⟨bit op ⟩] ⟨allkey ⟩

⟨bit op ⟩ → **IS** || **ISNT**

⟨allkey ⟩ → **ALLON** || **ALLOFF**

⟨comparator ⟩ → = || # || < || > || <= || >=

⟨ifnotst ⟩ → **IFNOT** ⟨condition ⟩ [⟨then ⟩] **NL**
{ ⟨statement ⟩ }* **END**

⟨condition ⟩ → ⟨cond ⟩ ⟨intval ⟩
|| ⟨allkey ⟩ ⟨plad list ⟩ ⟨intval ⟩

⟨killst ⟩ → **KILL** ⟨intval ⟩

⟨labelst ⟩ → **INT** : || **ID** :

⟨localst ⟩ → **LOCAL** ⟨local list ⟩ **NL**

⟨local list ⟩ → ⟨mode ⟩ **ID** [⟨initialise ⟩]
|| ⟨local list ⟩ , **ID** [⟨initialise ⟩]
|| ⟨local list ⟩ , ⟨mode ⟩ **ID** [⟨initialise ⟩]

⟨initialise ⟩ → = (⟨constant list ⟩)

⟨constant list ⟩ → ⟨constant ⟩
|| ⟨constant list ⟩ , ⟨constant ⟩

⟨nlst ⟩ → **NEWLINES** ⟨intval ⟩

⟨opst ⟩ → ⟨opkey ⟩ ⟨plad list ⟩

⟨opkey ⟩ → **OPEN** || **CLOSE**

⟨opchkst ⟩ → ⟨opchkey ⟩ ⟨plad list ⟩

⟨opchkey ⟩ → **OPENCK** || **CLOSECK**

⟨printst ⟩ → **PRINT** ⟨print item ⟩ [**INT**]

⟨print item ⟩ → ⟨variable ⟩
|| ⟨local var ⟩
|| ⟨key variable ⟩

⟨readkeyst ⟩ → **READKEY** ⟨intval write ⟩

⟨readpladst ⟩ → **READPLAD** ⟨variable ⟩ [⟨variable ⟩]

⟨readst ⟩ → **READ** ⟨variable ⟩
    || **READ** ⟨local var ⟩

⟨repeatst ⟩ → **REPEAT** ⟨exp ⟩ **NL** { ⟨statement ⟩ }* **FINISH**

⟨rtlst ⟩ → **RTL** ⟨intval ⟩

⟨setholdst ⟩ → **SETHOLD** ⟨intval ⟩

⟨setst ⟩ → **SET** ⟨variable ⟩ **=** ⟨exp ⟩
    || **SET PLAD** . ⟨selector ⟩ **=** ⟨exp ⟩

⟨setprivst ⟩ → **SETPRIV** ⟨intval ⟩ ⟨intval ⟩

⟨setseqst ⟩ → **SETSEQ** ⟨int list ⟩ ⟨intval list ⟩

⟨int list ⟩ → **INT**
    || ⟨int list ⟩ , **INT**

⟨intval list ⟩ → ⟨intval ⟩
    || ⟨intval list ⟩ , ⟨intval ⟩

⟨spst ⟩ → **SPACES** ⟨intval ⟩

⟨ssst ⟩ → **SSHOT** ⟨intval ⟩

⟨startst ⟩ → **START** ⟨intval ⟩ **ID** [⟨interface var ⟩] [⟨plist list ⟩]

⟨stepst ⟩ → **STEPON** ⟨intval ⟩ ⟨intval ⟩

⟨settagst ⟩ → **SETTAG** ⟨intval ⟩ ⟨intval ⟩

⟨textst ⟩ → **TEXT** ⟨messlist ⟩

⟨messlist ⟩ → ⟨mess ⟩ || ⟨messlist ⟩ , ⟨mess ⟩

⟨timeoutst ⟩ → **TIMEOUT**
    || **TIMEOUT HOLD** ⟨intval ⟩
    || **TIMEOUT CONT** ⟨intval ⟩
    || **TIMEOUT** ⟨intval ⟩ ⟨intval ⟩

⟨todst ⟩ → **TOD** ⟨intval write ⟩

⟨unchkst ⟩ → **UNCHECK** ⟨intval ⟩ [⟨plad list ⟩]

⟨plad list ⟩ → ⟨plad item ⟩
    || ⟨plad list ⟩ , ⟨plad item ⟩

⟨plad item ⟩ → **PLAD** || ⟨variable ⟩

⟨vdust ⟩ → **VDU** ⟨intval ⟩ ⟨intval ⟩

⟨whilest ⟩ → **WHILE** ⟨complex cond ⟩ **DO NL**
    { ⟨statement ⟩ }* **ENDWHILE**

⟨writest ⟩ → **WRITE INT** ⟨write list ⟩

⟨write list ⟩ → ⟨write item ⟩
    || ⟨write list ⟩ , ⟨write item ⟩

⟨write item ⟩ → ⟨mess constant ⟩
    || **PLAD** [⟨selector ⟩]
    || ⟨local var ⟩
    || ⟨key variable ⟩
    || ⟨variable ⟩

⟨xcallst ⟩ → **XCALL ID**

Appendix C

VIRTUAL MACHINE DEFINITION

Ideally an intermediate language serves as the "machine language" of some virtual machine. A clear understanding of this virtual machine would facilitate the implementation of an interpreter for the intermediate language. In the case of the CASL intermediate language, however, it seems that no such virtual machine has been defined. The following is a description of the virtual machine as constructed from the list of opcodes and by examining the code of an existing interpreter

Registers

The virtual machine consists of the registers listed in Figure C.1.

Memory

The memory of the virtual machine consists of two separate partitions, the sequence logic partition and the data partition. This allows for both sequence logic and data to be replaceable independently. Thus a particular sequence or data block may be marked and extracted and the modified sequence or data definition replaces the old definition. The memory may be considered as having variable length segments, with each segment

Name	Description
A	A main accumulator
B	A secondary accumulator
C	An accumulator used for skips etc
P	Program counter
PARAM	Stores the first operand of the opcode
Q	Stores the second operand
PSAVE	Stores the value of the program counter between phases
CC	Stores a pointer to the current cell
CSTAT	Stores the current status of a sequence
CS	Stores a pointer to the current sequence
CP	Stores a pointer to the current PLIST
SUBS	Subscript access register
BASE	Store the current PLIST
SEQREG(1)	Sequence register 1 for job number
SEQREG(2)	Sequence register 2 for tag number
SEQREG(3)	Sequence register 3 for sequence number
SEQREG(4)	Sequence register 4 for level of interest
SEQREG(5)	Sequence register 5 for job status
SEQREG(6)	Sequence register 6 for current level
SEQREG(7)	Sequence register 7 for last label passed
SEQREG(8)	Sequence register 8 for FUTHOLD label number
SEQREG(9)	Sequence register 9 for PLIST parameter number
SEQREG(10)	Sequence register 10 for PLIST parameter number
SEQREG(11)	Sequence register 11 for PLIST parameter number
SEQREG(12)	Sequence register 12 for PLIST parameter number

Figure C.1: Registers in the CASL virtual machine

indicating the length of the segment and its availability for use. Each PLIST (parameter list) is a variable length data block which is stored in a segment in the data partition. Data items within a parameter list are accessed by indexing, using the correct offset within the data block. Thus a method of double indexing or nested indexing is used in order to access the data items. A similar method of double indexing is used to access the opcodes in a sequence loaded into the sequence logic partition.

Opcodes

The intermediate language opcodes are presented in numerical order, with the opcode number represented by octal digits. The layout of each opcode is as follows:

- **OPCODE** **Symbolic_name**

  P Value of operand 1 if present

  Q Value of operand 2 if present

  DESCRIPTION OF THE OPCODE

- **000** **Illegal Opcode**

- **001** **Resetpc(P, Q)**

  P High word of the new program counter

  Q Low word of the new program counter

  This instruction changes the value of the program counter(PC) of the currently executing job. The 32-bit address is formed by joining the two operands as follows ((P SLL 8) LOR Q). The next instruction will be executed from the current sequence at this pc value.

- **002** **Resetpcp(P, Q)**

  P High word of the new program counter

  Q Low word of the new program counter

Same as above except that it forces an end-of-phase in order to give other jobs a chance to execute.

- **003** **Settimeout(P, Q)**

P High word value of an address

Q Low word value of an address

The Settimeout instruction forms part of intermediate code generated as a result of a CASL **IFNOT** statement. The **IFNOT** is a combination of an **IF** and a **TIMEOUT** statement. The two operands are joined to form a 32-bit address (i.e. (P SLL 8) LOR Q). After the specified number of seconds has expired, a Goto is executed to this address in the sequence code.

004 **Closearray(P, Q)**

P Item number of the plad array in the PLIST

Q The dimension of the array.

This instruction is used to perform a **CLOSE** or **CLOSECK** operation on an array of up to 16 **PLAD** values, stored as an array in a PLIST. It loops from item (P + 1) to item (P + Q) in the PLIST. It retrieves each **PLAD** item from the PLIST, then calls a function to perform the action.

- **005** **Openarray(P, Q)**

P Item number of the plad array in the PLIST

Q The dimension of the array

This instruction is used to perform an **OPEN** or **OPENCK** operation on an array of up to 16 **PLAD** items, stored as an array in a PLIST. This instruction loops from item (P + 1) to item (P + Q) in the PLIST. It retrieves each **PLAD** item from the PLIST, then calls a function to perform the action.

- **006** **Bitarray(P, Q)**

P Item number of the plad array in the PLIST

Q The dimension of the array

This instruction is used to read up to 16 digital tags, as bits, packing them into the A register. The tags are specified as an array of PLAD items, stored as an array in a PLIST. This instruction loops from item (P + 1) to item (P + Q) in the PLIST. It retrieves each PLAD item from the PLIST, then calls a function to get the bit value. The bits are shifted up during packing, with the last bit read in the least significant bit position.

- **007** **Storepararray(P, Q)**

P Item number in PLIST

Q Dimension of the array

When a PLIST parameter to a sequence in a CASL **CALL** or **START** statement is initialised, it is necessary to store the values in the PLIST. If an array of items are initialised with the same value then this opcode takes care of the initialisation of all these items.

- **010** **Getcell(P, Q)**

P Sequence number

Q Number of formal parameters

Getcell is used as part of the **START**, **GETSEQ** and **CALL** instruction sequences. A function is called, to check that a free cell is available. It also starts to load information into the local cell. Finally a procedure is used by **CALL** to link in the new cell as the current cell or in the case of the **START** or **SETSEQ** a function is called to make the new cell the base cell of the new job.

- **011** **SetBconst(P, Q)**

P High word value of the constant

Q Low word value of the constant

Register B, is set to a constant formed by joining the two operands.

B = ((P SLL 8) LOR Q)

- **012** **SetAconst(P, Q)**

P High word value of the constant

Q Low word value of the constant

Register A, is set to a constant formed by joining the two operands.

A = ((P SLL 8) LOR Q)

- **013** **Ckarray(P, Q)**

P Item number of plad array in the PLIST

Q Dimension of the array

Specifies the range of PLAD items to be checked in a PLIST. This opcode is generated by the CASL CHECK statement.

- **014** **XCall**

P High word of the address

Q Low word of the address

XCall is used to call an externally defined procedure. The loader is responsible for inserting the address of the procedure into the operands.

- **015** **Illegal opcode**

⋮

- **017** **Illegal opcode**

- **020** **Checkpoint**

This instruction initialises the interpreter logic for the series of instructions resulting from a CLOSECK or OPENCK statement the CASL source code.

- **021** **Check**

Controls the phases of the OPENCK or CLOSECK CASL statements.

- **022** **SetDelay**

  This instruction sets the current job into a DELAYED state. The duration is specified in seconds, which has been stored in the B register.

- **023** **SetDelayM**

  This instruction sets the current job into a DELAYED state. The duration is specified in minutes, which has been stored in the B register.

- **024** **Close**

  Closes a specified PLAD. Close is used as a result of the CASL CLOSE and CLOSECK statements.

- **025** **Open**

  Opens a specified PLAD. Open is used as a result of the CASL OPEN and OPENCK statements.

- **026** **Indigit**

  The value of the PLAD attribute twenty three (OPN) is loaded into the B register. It is used to test if the plad is open or closed (on or off).

- **027** **Setseq**

  Setseq is a generalised method of starting a job with the various job parameters initialised to values normally inaccessible to the START instruction. The opcode is a result of the CASL GETSEQ statement.

- **030** **Illegal opcode**

  ⋮

- **034** **Illegal opcode**

- **035** **Hold**

  The job's state is changed to HELD, thus stopping the execution of the job and causing a message to be produced on the system printer.

- **036** **Pause**

  The pause statement terminates the current phase.

- **037** **Endseq**

  Causes the execution of a sequence to be terminated and any timeout set for that sequence is cancelled. If it is the base sequence of a job, the job is killed.

- **040** **GotoI**

  Performs an indirect Goto. The label to which it jumps is stored in the B register, which is only determined at run-time.

- **041** **SetTimeoutHold**

  The timeout bit is set in the status byte for the current, job then after the specified time has elapsed, the job is put on hold.

- **042** **SetTimeoutCont**

  After a specified time has elapsed, the job is made ACTIVE. An error occurs if the job was not in the HELD state.

- **043** **Getscb**

  Getscb, gets a new sequence control block (should be called job control block). It checks and finds an unused job number. The opcode is produced every time a new job is created.

- **044** **Call**

It is the final instruction in the generalised **CALL** logic. It is preceded by Getcell as well as an instruction to load the PLIST parameters either directly into the cell or into the parameter pool block.

- **045 Start**

  Start is used to activate a new job.

- **046 Setparabase**

  Setparabase sets up parameters as part of a **START** or **CALL** instruction sequence. It checks the validity of the PLIST number. If ok, they then set up BASE, in order to possibly initialise values in the PLIST, and store the PLIST number into the parameter list for the new cell.

- **047 Clseqreg**

  This instruction clears all the sequence registers.

- **050 Me**

  This instruction copies the key variable **ME** (current job number) into register B. At the same time it copies B into the subscript register SUBS allowing it to be used as an index into the list of PLIST items.

- **051 Tag**

  This instruction copies the key variable **TAG** (tag value for the current job) into register B. At the same time it copies B into the subscript register SUBS allowing it to be used as an index into the list of PLIST items.

- **052 Priv**

  This instruction copies the key variable **PRIV** (the privilege of the current job) into register B. At the same time it copies B into the subscript register SUBS allowing it to be used as an index into the list of PLIST items.

- **053** **Error**

  This instruction copies the key variable ERROR (error number of the last error detected) into register B. At the same time it copies B into the subscript register SUBS allowing it to be used as an index into the list of PLIST items. At the same time, ERROR is cleared for the current job.

- **054** **Ck**

  Depending on the type of check as set by the ckflag opcode, Ck either unchecks a PLAD list or checks a PLAD list for a DIVERT ME or for a DIVERT CONT CASL statement.

- **055** **Pladpart2**

  No longer supported.

- **056** **Pladpart3**

  No longer supported.

- **057** **Flagerrorplads**

  Outputs all the invalid PLADs in a CASL IFNOT statement.

- **060** **Divert**

  This instruction does a DIVERT job control operation on the job number specified in register B. Register A specifies the label number for the divert.

- **061** **Continue**

  If the job is not on hold an error occurs, otherwise the job becomes ACTIVE. The job number is stored in register B.

- **062** **Sethold**

This instruction does a hold job control operation on the job number specified in register B. The HSB (held status bit) is set.

- **063** **Kill**

  This instruction kills the job whose number is specified in register B.

- **064** **SShot**

  It should un-hold a job, execute one instruction and then place the job back on hold again. The job number is specified in register B.

- **065** **Futhold**

  Sets a future hold. When the job arrives at a specified label in a particular sequence it is put on hold. The sequence, job and label numbers are found in the A, B and C registers.

- **066** **Settag**

  Initialises the tag number of a job. The job number is stored in register B while the new tag number is retrieved from register A.

- **067** **Stepon**

  Performs a manual divert operation.

- **070** **Setpriv**

  Sets the privilege of a job, specified in the B register.

- **071** **SetLoopcnt**

  The result of an expression in the CASL **REPEAT** statement is stored in register A. This value is then used to set the loop counter in the current cell.

- **072** **Interbase**

  The BASE register is set to the interface PLIST number for the sequence. This register is initialised when an interface variable is used in the sequence logic

- **073** **Localbase**

  The BASE register is set to the current sequence's local PLIST number. The register is initialised when a local PLIST variable is used in the sequence logic.

- **074** **Illegal opcode**

  ⋮

- **077** **Illegal opcode**

- **100** **Label(P)**

  P Label number

  After storing the label number in the current cell it checks whether any future hold conditions apply.

- **101** **Setparabase(P)**

  P PLIST number

  It is used to set up parameters as part of a **START** or **CALL** instruction sequence. The actual PLIST number is associated with the PLIST parameter name declared for the sequence.

- **102** **Interparabase(P)**

  P Called sequence number

  A **CALL** or **START** statement has the ability to pass variables to the invoked sequence. These variables are stored as a PLIST, and the PLIST is referred to as an Interface PLIST. In the **CALL** and **START** statement the variables may be initialised, therefore

the values must be stored. The opcode instructs the interpreter which PLIST is being initialised.

- **103** **SetAbyte(P)**

  P Word value

  Initialises the A register to the 16-bit word value.

- **104** **Call0(P)**

  P Sequence number

  The opcode is used for a CASL `CALL` statement when there are no parameters being passed to the calling sequence.

- **105** **Start0(P)**

  P Sequence number

  The opcode is used for a CASL `START` statement when there are no parameters being passed to the sequence, to be started.

- **106** **Setmess(P)**

  P String number

  The string number is joined with the jobs string pool number to form a 32-bit address. This address is then stored in the B register.

- **107** **Illegal opcode**

- **110** **Startwrite(P)**

  P Write priority number

  This is the first instruction of the sequence generated by the CASL `WRITE` statement. It initialises variables for the upcoming output statements.

- **111** **Printint(P)**

P Print format width

Prints the integer value found in the B register.

- **112** **Printnum(P)**

P Print format width

Prints the real value found in the B register.

- **113** **Printbits(P)**

P Print format width

Prints the bits value found in the B register in octal format.

- **114** **Printplad(P)**

P Print format width

Prints out the tag name of a plad variable found in the B register.

- **115** **Illegal opcode**

⋮

- **117** **Illegal opcode**

- **120** **Vdu**

It allows device specific VDU operations to be performed. Registers A and B specify the type of operation.

- **121** **Writeint**

Writes the integer value stored in the B register to the system printer.

- **122** **Writenum**

Writes the real number value stored in the B register to the system printer.

- **123 Writebits**

  Writes the bits value stored in the B register, in octal format on the system printer.

- **124 Writeplad**

  Writes the plad value stored in the B register to the system printer.

- **125 Writemess**

  Writes the string constant in the string pool pointed to by the address in the B register to the system printer.

- **126 Spaces**

  This instruction writes out a number of spaces specified in register B.

- **127 Newlines**

  This instruction writes out a number of newlines specified in register B.

- **130 Illegal opcode**

- **131 Readint**

  This instruction reads an integer value and stores the result in the A register.

- **132 Readnum**

  This instruction reads a real number and stores the result in the A register.

- **133 Illegal opcode**

- **134 Readplad**

  Reads a plant address value into the A register.

- **135** **Illegal opcode**

- **136** **Readkey**

  This instruction is used to interpret responses from an operator, via the OCP. It returns a "raw" keycode as the bottom 8 bits of an integer value, returned in the A register.

- **137** **Pladpart1**

  No longer supported.

- **140** **Endwrite**

  This is the last instruction generated by the CASL statement **WRITE**. If a **WRITE** 255 is used it clears the **MESS** variable assignment flag. In the case of write to a physical device, it finishes with a newline. To the OCP topic queue it finishes with a flush.

- **141** **Devin**

  The input stream is initialised to the value stored in the B register.

- **142** **Devout**

  The output stream is initialised to the value stored in the B register.

- **143** **Illegal opcode**

  ⋮

- **177** **Illegal opcode**

- **200** **Setatt(P)**

  P PLAD attribute value

  The plad attribute P, of the plad in register B is initialised to the value stored in the A register.

- **201** **Fetchatt(P)**

  P PLAD attribute value

  The plad in register B has attribute P's value retrieved from the database and stored in register B.

- **202** **Ckflag(P)**

  P Check flag number

  The instruction determines how the event check list is to be manipulated. The flag number may have a value between zero and four. Zero specifies all plads to be checked, one is unused, while two specifies the plad list to be unchecked, three does a check divert operation and four does a check continue operation.

- **203** **SetBbyte(P)**

  P Word value

  This instruction copies the 16-bit word value to the B register.

- **204** **Addtobase(P)**

  P Base PLIST number

  When at compile time the PLIST number cannot be determined due to a variable specifying the PLISTs instantiation number, this opcode is used. The operand value is the lowest PLIST number associated with that name in the **PLISTNUM** section of the CASL source program.

- **205** **Setbitatt(P)**

  P Bits PLAD attribute value

  A bits value is stored in the database for the plad found in the B register and the attribute value is passed as an operand. The bits value to be stored is found in the A register.

- **206** **Fetchbitatt(P)**

P Bits PLAD attribute value

A bits plad attribute value is retrieved from the database and stored in the B register.

- **207** **Storepara(P)**

P Item number in a PLIST

The value in the A register is stored in the item specified by the operand in the current PLIST .

- **210** **GotoD(P)**

P Label number

Performs a jump to the address in the current sequence corresponding to the label number.

- **211** **SetLoop(P)**

P Integer value

The loop counter is initialised to the integer value.

- **212** **Setseqreg(P)**

P Sequence register number

This instruction is one of a series generated in order to set up sequence registers values. If the sequence number is valid then the value is copied from register B to the sequence register number specified by the operand.

- **213** **Getseq(P)**

P Sequence number

GETSEQ is a generalised method for locating useful information about a sequence.

- **214** **Setbasepara(P)**

P Parameter number

The PLIST corresponding to the PLIST parameter number is stored in the BASE register.

- **215** **Setbase(P)**

  P PLIST number

  The PLIST number is stored in the BASE register.

- **216** **Flaggedcond(P)**

  P Number of PLADs

  Sets a flag for certain conditions.

- **217** **SetTimeoutlab(P)**

  P Label number

  It sets a timeout and saves the conditions in the current cell. After the timeout has expired, the program jumps to a specified label. The duration of the timeout is retrieved from register B and the label number from the operand.

- **220** **BittoA**

  Bit manipulation of the A and B registers with the result placed in the A register.

- **221** **Decrement**

  Decrease the loop counter by one, if it is zero reset the program counter and jump to the next instruction.

- **222** **Illegal opcode**

- **223** **Btoplad**

  Stores the B register into the lastplad field in the job control block.

- **224 Atopart1**

  No longer supported.

- **225 Atopart2**

  No longer supported.

- **226 Formplad**

  No longer supported.

- **227 Setbasepara1**

  Retrieves the first PLIST from the parameter pool and stores it in the current PLIST field of the job control block.

- **230 Addint**

  Adds the integer values in the A and the B registers and places the result in the A register.

- **231 Subint**

  Subtracts the B from the A register and stores the result in the A register. The values in the A and B registers are integers.

- **232 Multint**

  The integer values in the A and B registers are multiplied together and the result is stored in the A register.

- **233 Divint**

  The integer value in the A register is divided by the integer value in the B register, with the result stored in the A register.

- **234** **Addintdump**

  Adds the integer values in the PSAVE and A registers, with the result stored in the A register.

- **235** **Subintdump**

  The integer value in the A register is subtracted from the PSAVE register, with the result stored in the A register.

- **236** **Neg**

  Negates the B register.

- **237** **Illegal opcode**

- **240** **Addnum**

  The real numbers in the A and B registers are added to each other with the result stored in the A register.

- **241** **Subnum**

  The real number in the B register is subtracted from the value of the A register.

- **242** **Multnum**

  The real numbers in the A and the B registers are multiplied and the result is stored in the A register.

- **243** **Divnum**

  The A register is divided by the real number in the B register, with the result stored in the A register.

- **244** **Addnumdump**

The real number in the PSAVE and A registers are added to each other and saved in the A register.

- **245 Subnumdump**

  The A register real number is subtracted from the PSAVE register, with the result stored in the A register.

- **246 Fix**

  Converts the integer value in the B register to a real number.

- **247 Float**

  Converts the real number in the B register to an integer number.

- **250 TimetoAS**

  Stores the time in seconds since startup into the A register.

- **251 TimetoAM**

  Stores the time in minutes since startup into the A register.

- **252 Tod**

  Reads the current time of day from the system clock, and returns four decimal digits in the format HHMM to the A register.

- **253 Cleartimeout**

  Clears the active job's timeout at its current level.

- **254 Ident1**

  Ident1 is used to set up the current activity of the sequence. They store the string pool index to the strings which form the preamble to all standard messages output by interpreter.

- **255** **Ident2**

  Ident2 is used to set up the current activity of the sequence. They store the string pool index to the strings which form the preamble to all standard messages output by interpreter.

- **256** **Event**

  Sets the kth marker in the event flag list. Where the kth marker is found in the B register.

- **257** **RTL**

  Transfers control from the interpreter to a numbered RTL/2 procedure. When the RTL/2 procedure is finished control returns to the interpreter.

- **260** **Skip if =**

  If the C register is zero the next instruction (a Resetpc) is skipped.

- **261** **Skip if not equal**

  If the value in the C register is not equal to zero the next instruction (a Resetpc) is skipped.

- **262** **Skip if <**

  If the value in the C register is less than zero the next instruction (a Resetpc) is skipped.

- **263** **Skip if >**

  If the value in the C register is greater than zero the next instruction (a Resetpc) is skipped.

- **264** **Skip if <=**

If the value in the C register is less than or equal to zero the next instruction (a Resetpc) is skipped.

- **265** **Skip if >=**

  If the value in the C register is greater than or equal to zero the next instruction (a Resetpc) is skipped.

- **266** **Illegal opcode**

- **267** **Bdump**

  Stores the value in the B register to the PSAVE register which stores intermediate results.

- **270** **SetCint**

  The C register is set after it is compared to the integer value in the A register. If the C register is greater than the A register the C register is set to one. If the value is less than the A register the C register is set to negative one. If the values are equal the C register is set to zero.

- **271** **SetCnum**

  The C register is set in a similar way as in SetCint. Only here the contents of the A and C registers are real numbers.

- **272** **SetCbits**

  Performs an exclusive OR operation on the C and A registers. The result is placed into register C.

- **273** **AtoC**

  Copies the A register to the C register.

- **274** **Dump**

  Stores the value found in the A register into the PSAVE register.

- **275** **BtoA**

  Copies the B register to the A register.

- **276** **ClrA**

  Clears the A register.

- **277** **ClrB**

  Clears the B register.

- **300** **Storesubs(P)**

  P Item number in the PLIST

  It uses the SUBS register plus the item number to access the PLIST's item. And stores the value in A into that address.

- **301** **Fetchsubs(P)**

  P Item number in the PLIST

  Uses the SUBS register to determine the item's location. Then stores the value pointed to by that location in register B.

- **302** **Setbasevarsubs(P)**

  P Item number in the PLIST

  Used to set BASE for a PLIST whose instantiation number is a subscripted integer variable.

- **303** **Setsubsvarsubs(P)**

  P Item number in the PLIST

Stores the PLIST item, which is found by using indirect with displacement addressing, into the SUBS register.

- **304** **Sublogintsubs(P)**

  P Item number in the PLIST

  Subtracts the PLIST item from the A register and stores the result into the PLIST item. The PLIST item is found by using indirect with displacement addressing.

- **305** **Storebitsubs(P)**

  P Item number in the PLIST

  Sets or clears bits in the PLIST item. The PLIST item is found by indirect with displacement addressing.

- **306** **Fetchbitsubs(P)**

  P Item number in the PLIST

  Fetches bits from the PLIST item and places them in the B register. The PLIST item is found by using indirect with displacement addressing.

- **307** **Illegal opcode**

- **310** **Store(P)**

  P Item number in the PLIST

  Stores the A register into the PLIST item.

- **311** **Fetch(P)**

  P Item number in the PLIST

  Fetches the PLIST item, and stores the result in register B.

- **312** **Setbasevar(P)**

  P Item number in the PLIST

Stores the PLIST item, which is found by indirect addressing, into the BASE register.

- **313** **Setsubsvar(P)**

  P Item number in the PLIST

  Stores the PLIST item, which is found by indirect addressing, into the SUBS register.

- **314** **Sublogint(P)**

  P Item number in the PLIST

  Subtracts the PLIST item from the A register and stores the result into the PLIST item.

- **315** **Storebit(P)**

  P Item number in the PLIST

  Sets or clears bits in the PLIST item. The PLIST item is found by using indirect addressing.

- **316** **Fetchbit(P)**

  P Item number in the PLIST

  Fetches bits from the PLIST item and places them in the B register. The PLIST item is found by using indirect addressing.

- **317** **Illegal opcode**

- **320** **StoreX1**

  Store the value in the A register into the job's local working variable `X1`.

- **321** **FetchX1**

  Fetch the job's local working variable `X1` and stores the value in the B register.

- **322** **SetbaseX1**

  The BASE register is set to the job's local working variable X1.

- **323** **SetsubsX1**

  The SUBS register is set to the job's local working variable X1.

- **324** **SublogintX1**

  Subtracts the job's local working variable X1, from the A register and stores the result into the X1.

- **325** **Illegal opcode**

  ⋮

- **327** **Illegal opcode**

- **330** **StoreX2**

  Store the value in the A register into the job's local working variable X2.

- **331** **FetchX2**

  Fetch the job's local working variable X2 and stores the value in the B register.

- **332** **SetbaseX2**

  The BASE register is set to the job's local working variable X2.

- **333** **SetsubsX2**

  The SUBS register is set to the job's local working variable X2.

- **334** **SublogintX2**

Subtracts the job's local working variable X2, from the A register and stores the result into the X2.

- **335** **Illegal opcode**

  ⋮

- **337** **Illegal opcode**

- **340** **StoreX3**

  Store the value in the A register into the job's local working variable X3.

- **341** **FetchX3**

  Fetch the job's local working variable X3 and stores the value in the B register.

- **342** **SetbaseX3**

  The BASE register is set to the job's local working variable X3.

- **343** **SetsubsX3**

  The SUBS register is set to the job's local working variable X3.

- **344** **SublogintX3**

  Subtracts the job's local working variable X3, from the A register and stores the result into the X3.

- **345** **Illegal opcode**

  ⋮

- **347** **Illegal opcode**

- **350** **StoreX4**

  Store the value in the A register into the job's local working variable **X4**.

- **351** **FetchX4**

  Fetch the job's local working variable **X4** and stores the value in the B register.

- **352** **SetbaseX4**

  The BASE register is set to the job's local working variable **X4**.

- **353** **SetsubsX4**

  The SUBS register is set to the job's local working variable **X4**.

- **354** **SublogintX4**

  Subtracts the job's local working variable **X4**, from the A register and stores the result into the **X4**.

- **355** **Illegal opcode**

  ⋮

- **377** **Illegal opcode**

Appendix D

CASL RESERVED WORDS

ALLOFF
ALLON
AND
BIN
BITS
CALL
CHECK
CLOSE
CLOSECK
CONT
CONTINUE
DATA
DELAYM
DELAYS
DEVIN
DEVOUT
DIVERT
DO
ELSE
ELSEIF
END
ENDSEQ
ENDWHILE
ERROR
EVENT
FINISH
FOR
FUTHOLD
GETSEQ
GOTO
HEX
HOLD
IDENT1/2
IF
IFNOT
INT
IS
ISNT
KILL
LET
LOCAL
ME
MELTIME
MESS
MTIME
NEWLINES
NEXT
NUM
OCT
OPEN
OPENCK
OR
PAUSE
PLAD
PLIST
PLISTDEF
PLISTNUM
PRINT
PRIV
READ
READKEY
READ PLAD
REPEAT
RETURN
RTL
SEQNUM
SEQUENCE
SELTIME
SET
SETHOLD
SETPRIV
SETSEQ
SETTAG
SPACES
SSHOT
START
STEP
STEPON
STIME
TAG
TEXT
THEN
TIMEOUT
TITLE
TO
TOD
UNCHECK
VDU
WHILE
WRITE
X1
X2
X3
X4
XCALL

Bibliography

[1] Abraxas Software, Inc., 7033 SW Macadam Ave., Portland, Oregon, 97219, USA. *PCYACC.*

[2] AECI Process Computing, P.O. Box 1823 Halfway House South Africa. *CYGNUS PROCESS AND MANAGEMENT SOFTWARE.*

[3] AECI Process Computing, P.O. Box 1823 Halfway House South Africa. *CASL USER MANUAL*, 8 edition, Jan 1987.

[4] A.V. Aho and S.C. Johnson. LR parsing. *Computing Surveys*, 6(2):99–124, 1974.

[5] A.V. Aho and J.D. Ullman. *Principles of Compiler Design.* Addison-Wesley, Massachusetts, 1977.

[6] J.G.P. Barnes. Real time languages for process control. *Computer Journal*, 15:15–17, 1972.

[7] J.G.P. Barnes. *RTL/2 Design and Philosophy.* Heyden and Son Ltd., Great Britian, 1976.

[8] J.G.P. Barnes and R.J. Long. Process control systems using RTL/2. In *Software for Control*, pages 75–80, 1973.

[9] Y. Bashan and M. Handelsman. Microcomputer-based control of a batch process. *Chemical Engineering*, 94(1):123–125, 1987.

[10] C. Charalambous and A.J. Conning. Distributed sequence control utilising a high level sequencing language in conjunction with PLCs. In I.M. Macleod and A.D. Heher, editors, *Software for Computer Control 1988. Proceedings of the fifth IFAC/IFIP Symposium*, pages 27–36, 1988.

[11] T. Cobb. Private Communication. Malkins Software, 129 Hillcrest Drive, Ontario, L3Y 4V8, Canada.

[12] L. Darda, P. Gricuk, and J. Kolodziejski. Interpretive language for sequence control of chemical processes. In M. Novak, editor, *Software for Computer Control 1979. Proceedings of the Second IFAC/IFIP Symposium*, pages 207–210, 1979.

[13] W.M. Dehnert and V.H. Hasse. High-level language structures for distributed real-time programming. In M. Novak, editor, *Software for Computer Control 1979. Proceedings of the Second IFAC/IFIP Symposium*, pages 189–194, 1979.

[14] R.W. Dehning. Private Communication. AECI Process Computing Research Associate.

[15] S.L Fihn and J.A. Nyquist. Specifying batch process control strategies: A structured approach. *Intech*, 29:57–63, 1982.

[16] IEE. *Software for Control.* IEE, 1973.

[17] B.W. Kernigham and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, 1978.

[18] M.A. Keyes, D.M. Norris, and J.G. Patella. A hybrid approach to batch control. In *Generic Control for Batch Manufacturing, Proceedings of the 16th Annual Advanced Control Conference.*, pages 91–96, 1990.

[19] E.J. Kompass, S.K. Whitlock, and T.J. Williams, editors. *Generic Control for Batch Manufacturing, Proceedings of the 16th Annual Advanced Control Conference.* Purdue Laboratory for Applied Industrial Control, Purdue University and Control Engineering Magizine, 1990.

[20] D.B. Leach. Specifying a batch-process control system. *Chemical Engineering*, 93(23):115–122, 1986.

[21] A. Lewis and G. Trainito. Implementation of a standard language for real-time distributed process control. In M. Novak, editor, *Software for Computer Control 1979. Proceedings of the Second IFAC/IFIP Symposium*, pages 203–206, 1979.

[22] I.M. MacLeod. The application of modern software engineering techniques in the development of a process control package. In G. Ferrate and E.A. Puente, editors, *Software for Computer Control 1982. Proceedings of the Third IFAC/IFIP Symposium*, pages 395–399, 1982.

[23] I.M. Macleod and A.D. Heher, editors. *Software for Computer Control 1988. Proceedings of the fifth IFAC/IFIP Symposium*, Oxford, 1988. IFAC/IFIP, Pergamon Press.

[24] L.E. Massey and J.J. McCarthy. Batch process automation for the "90. In *Generic Control for Batch Manufacturing, Proceedings of the 16th Annual Advanced Control Conference.*, pages 17–37, 1990.

[25] M. Novak, editor. *Software for Computer Control 1979. Proceedings of the Second IFAC/IFIP Symposium*, Oxford, 1979. IFAC/IFIP, Pergamon Press.

[26] Terrence W. Pratt. *Programming Languages: Design and Implementation.* Prentice Hall, New Jersey, second edition, 1984.

[27] S.E. Rubin. Supervisory batch control using personnal computers. In *Generic Control for Batch Manufacturing, Proceedings of the 16th Annual Advanced Control Conference.*, pages 103–106, 1990.

[28] A.T. Schreiner and Jr. H.G. Freidman. *Introduction to Compiler Construction with UNIX.* Prentice-Hall, Englewood Cliffs, 1985.

[29] W.T. Shaw. Structured design produces good batch control programs. *Control Engineering*, pages 72–76, Nov 1983.

[30] J.C. Ward and M.R. Scalera. Digital batch process control: Look at the software. *Intech*, 29:65–67, 1982.

[31] E.A. Warman. An engine test language for computer control of engine tests. In *Sofware for Control*, pages 137–142, 1973.