Pascal-FC
Version 5
User Guide for Sun Systems
Sun-PP

G.L. Davies
University of Bradford, UK

1. OPTIONAL AND IMPLEMENTATION-DEPENDENT FEATURES

1.1. Data types supported

1.1.1. The type `integer`

Integers have 32 bits of precision. The value of `maxint` is 2147483647, and the smallest integer that can be represented is `-maxint` (not `-maxint-1`).

1.1.2. The type `char`

The ASCII character set is used. Valid parameters to the `chr` function range from 0 to 127.

1.1.3. The type `real`

The standard type `real` is supported by this implementation. Real numbers are represented with approximately 14 significant digits. The approximate range of real numbers that can be represented is from -9.9999999999999E299 to +9.9999999999999E299. Any real number with an exponent more negative than -299 is considered zero.

1.1.4. The type `bitset`

These are sets of `0 .. 31.` The `write` and `writeln` procedures use *binary* notation for values of this type, with the most significant bit on the left. The `int` function treats a bitset as a 32-bit two's complement integer, and the `bits` function performs the inverse operation.

1.2. Based literals

The values for the base in a based literal are restricted to "2", "8" and "16". The compiler will flag an error if any other value is used.

1.3. Based output

The values for the base in a `write` or `writeln` call are restricted to "8" and "16". Any other value will cause *hexadecimal* format to be used (no error will be flagged). Based output is not applicable to the type `real`, nor can based output be used in conjunction with field-width specifications.

1.4. Based input

Integers may be input in binary, octal, hexadecimal or decimal forms in response to `read(ln)`. Decimal is the default, and the format for the other bases is exactly the same as that required in a Pascal-FC source program. The format *must not* be specified in the source program: input is interpreted appropriately at run time.

1.5. Processes and Process Scheduling

1.5.1. Choice from among executable processes

This implementation supports two different schedulers, which we call the *standard* and the *unfair*. In either case the standard procedure, `priority`, has the same effect as the `null` statement.

1.5.1.1. The standard scheduler

The standard scheduler is pre-emptive, which means that a process can be forced to relinquish the processor even though it is still executable. The scheduler selects a process to run by generating a random number. In addition, having determined which process to run, it generates a random number which determines how many instructions the process will be allowed to execute before it is next forced out of the processor. In practice, this number is small, so that process switches will occur frequently and no one process receives preferential treatment. This scheduler is particularly useful for revealing errors due to uncontrolled multiple update.

1.5.1.2. The unfair scheduler

The unfair scheduler uses a different rule to select a ready process for running. Rather than generating a random number, it always chooses the lowest-numbered executable process. Processes are numbered in the order in which they were activated in the concurrent statement. The unfair scheduler is not pre-emptive: once a process is running, it will continue to do so until either it blocks or runs to completion.

1.5.2. Maximum number of processes

There is no specific limitation on the number of processes that can be *declared* in a program, but no more than 100 may be activated in the concurrent statement. An attempt to exceed this number will cause the run-time system to abort the program.

1.6. Semaphores

In the event that there are several processes suspended on a semaphore when a `signal` is carried out, a random choice is made to determine which of them to unsuspend. This is not affected by the scheduler in use.

1.7. Monitors

As there is no notion of process priority in this implementation, all queues associated with monitors (including those on `condition` variables) are strictly FIFO. This is not affected by the scheduler in use.

1.8. Resources

The implementation of resources has the following characteristics:

1. Mutual exclusion at a resource boundary is implemented by a FIFO queue.

2. Barriers on guarded procedures are FIFO queues.
3. If there are several processes suspended on `true` barriers which could inherit mutual exclusion of the resource, a random choice is made to determine which is successful.

The above points apply regardless of which scheduler is used.

1.9. Ada-style rendezvous

Process entries have FIFO queues, which are not affected by the scheduler in use.

1.10. Selective Waiting

When there are several channel or accept alternatives which have open guards and pending calls, a random choice is made among them, except where the **`pri`** select is concerned. In the latter case, textual order determines the choice. The scheduler has no influence on the choice of rendezvous.

1.11. Timing Facilities

The system clock unit is one second. The `clock` function returns the time in seconds since the beginning of execution of the program.

The timing facilities provide only approximate timings: a process which executes the `sleep` procedure, or a **`select`** with a **`timeout`** alternative may be made runnable up to one second early.

1.12. Low-level Facilities

This implementation does not support mapping indicators in variable or entry declarations or offset indicators in record field declarations. There is in consequence no facility to manipulate hardware device registers or to handle interrupts. The type `bitset` is supported, and has been described earlier.

1.13. Restrictions imposed by compiler Internal Tables

Table 1.1 lists the relevant restrictions. Exceeding any of these limits, except the number of significant characters in an identifier, will result in a fatal error. In the case of an identifier, characters in excess of the limit are read but discarded.

Number of significant characters in an identifier	10
Maximum source line length (characters)	121
Size of identifier table (entries)	500
Size of block table (entries)	30
Size of string table (characters)	1500
Size of array table (entries)	30
Maximum depth of nesting of blocks	15
Maximum length of generated code (P-code instructions)	2000
Size of channel table (entires)	20
Maximum number of alternatives in a **`select`**	20
Maximum exported procedures in a monitor or resource	20
Size of real literal table (entires)	50
Size of enumeration type table (entires)	20
Maximum number of alternatives in a **`case`**	20
Maximum number of monitors or resources in a program	10

Table 1.1: Compiler Internal Table Constraints

2. COMPILING AND RUNNING A PROGRAM

2.1. Invoking the Compiler

The Pascal-FC system consists of separate compiler and interpreter programs. Figure 2.1 shows the system components and the files that they produce or use.

2.1.1. The `pfc` command

The `pfc` command can be used to compile and run the source program. The command has the form:

```
pfc [-uf] <sourcefile_name>
```

No file name conventions are enforced by the compiler, but it is recommended that you

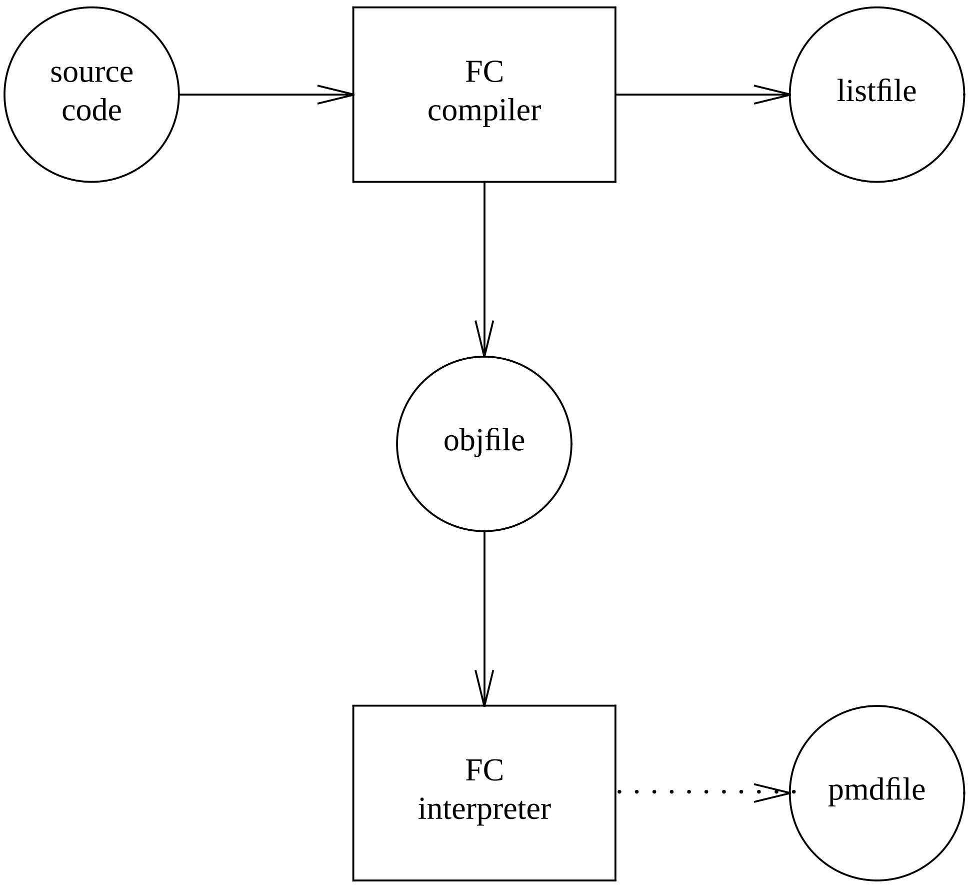

Figure 2.1: Files and Pascal-FC System Components

use a `.pfc` extension. The optional `-uf` flag causes the unfair scheduler to be used in running the program: the default is the standard scheduler.

The Pascal-FC program may be re-run any number of times without the need to restart the interpreter. The following message will be displayed:

```
Type r and RETURN to rerun
```

Typing "r" or "R" will cause the program to run again; any other character will terminate the interpreter.

Using the standard scheduler, re-running a program in this way will in general produce a different interleaving of processes. Note that there may be some indeterminacy even when using the unfair scheduler, because some scheduling decisions are still determined randomly (for example, when a `signal` is carried out on a semaphore on which several processes are blocked),

2.2. Input and output redirection

Output produced by the source program is normally sent to the screen. Normal Unix output redirection can be used when using the `pfc` command if a permanent copy of program output is required. However, redirection is not entirely satisfactory, because any prompts for input from the source program will also be redirected to the named file. So, indeed, will the invitation to "Type r to rerun" when the program has been executed.

Input redirection is also feasible, but the response to the invitation to re-run must be included with the input data if more than one run is required. In preparing the data file, the user should bear in mind that the interpreter code which manages the re-run facility has the form:

```
repeat
   runprog;
   writeln;
   writeln('Type r and RETURN to rerun');
   if eoln then readln;
      if eof then
         ch := 'x'
      else
         readln(ch);
   writeln;
until not (ch in ['r','R'])
```

Hence, the Pascal-FC source program may or may not need an explicit `readln` to consume all the data prior to the re-run response. If the file is intended to contain only sufficient data for one run, then the response to the prompt need not be included, because the interpreter will detect the end-of-file condition and exit normally.

2.3. The run-time post-mortem dump

Pascal-FC provides a post-mortem dump when a program generates a run-time error. The dump is sent to the `pmdfile`. No file with this name is produced if the program terminates normally.

The post-mortem dump is intended to be self-explanatory, and it will be introduced here by means of examples, with a small number of supporting comments.

2.3.1. Example: accidental omission of a terminate alternative

The following is an attempt at a buffered solution to the producer-consumer problem using channels. However, there is no **`terminate`** alternative in the `buffer` process.

```
program pcon6a;

(* buffered producer-consumer with channel rendezvous *)


var
  inp, out: channel of char;


process buffer;


const
  buffmax = 4;

var
   store: array[0..buffmax] of char;
   nextin, nextout, count: integer;
```

```
begin
   nextin := 0;
   nextout := 0;
   count := 0;
   repeat
      select
      when count <> 0 =>
         out ! store[nextout];
         count := count - 1;
         nextout := (nextout + 1) mod (buffmax + 1);
      or
      when count < buffmax =>
         inp ? store[nextin];
         count := count + 1;
         nextin := (nextin + 1) mod (buffmax + 1 );
      end  (* select *)
   forever
end;  (* buffer *)

process producer;

var
   local: char;

begin
   for local := 'a' to 'z' do
      inp ! local
end;  (* producer *)


process consumer;

var
   local: char;

begin
   repeat
      out ? local;
      write(local);
   until local = 'z';
   writeln
end;  (* consumer *)
```

```
begin
   cobegin
      producer;
      consumer;
      buffer
   coend
end.
```

When the program was run, the following appeared on the screen:

```
- Pascal-FC for Unix Systems -
- Compiler Version 5.1 -

G L Davies  &  A Burns, University of Bradford


Compiling pcon6a     ...
Compilation complete

- Interpreter Version 5.1 -
Program pcon6a    ...  execution begins ...


abcdefghijklmnopqrstuvwxyz

Abnormal halt in process buffer with pc = 68
Reason:   deadlock



See pmdfile for post-mortem report


Type r and RETURN to rerun
```

The contents of `pmdfile` were as follows:

```
Pascal-FC post-mortem report on pcon6a
- Interpreter Version 5.1 -
Abnormal halt in process buffer with pc = 68
Reason:   deadlock
```

```
----------
Main program

Status:  awaiting process termination


----------
Process producer

Status:  terminated


----------
Process consumer

Status:  terminated


----------
Process buffer

Status:  active
pc = 68

Process suspended on:

inp (channel)



==========
Global variables

(None)
```

Two points should be borne in mind when interpreting the post-mortem dump:

(1) The approximate position in the Pascal-FC source code where a process has been aborted can be found by using the *program counter* values printed in the associated `listfile`. The value of `pc` given in the post-mortem dump is the *next* instruction which would have been executed.

(2) Only *non-structured* variables of types `integer`, `real`, `char`, `boolean` and `semaphore` will be listed in the *global variables* section of the report.

2.3.2. Example: rendezvous that can never occur

Finally, here is a second example, together with the post-mortem dump produced.

```
program pmdtest5;

const
   nchans = 5;

var
   chanarray: array[1..nchans] of channel of synchronous;


process p;

var
   loop: integer;

begin
   select
      for loop := 1 to nchans replicate
         chanarray[loop] ? any
   end
end;


begin
   cobegin
      p
   coend
end.
```

```
Pascal-FC post-mortem report on pmdtest5
- Interpreter Version 5.1 -
Abnormal halt in process p with pc = 21
Reason:   deadlock
```

```
----------
Main program

Status:  awaiting process termination


----------
Process p

Status:  active
pc = 21

Process suspended on:

chanarray[1] (channel)
chanarray[2] (channel)
chanarray[3] (channel)
chanarray[4] (channel)
chanarray[5] (channel)



==========
Global variables

(None)
```

APPENDIX A - THE COMPILER LISTING FILE

1. Form of the listing file

The contents of the listing file fall into the following sections:

(1) the source code listing itself;

(2) the symbol table generated from the program;

(3) the generated code.

Only the first of these sections will be of any interest to the majority of users. Hence, the symbol table and generated code have been omitted from the example in the following section.

2. An example

```
- Pascal-FC for Unix systems -
- Compiler Version 5.1 -

G L Davies  &  A Burns, University of Bradford

Compiler listing for file gardens1.pfc


    1      0 program gardens1;
    2      0
    3      0 (* first attempt to solve the
    4      0    ornamental gardens problem *)
    5      0
    6      0 var
    7      0    count: integer;
    8      0

    9      0 process turnstile1;
   10      0
   11      0 var
   12      0    loop: integer;
   13      0 begin
   14      0    for loop := 1 to 20 do
   15      4       count := count + 1
   16      7 end;  (* turnstile1 *)
   17     10
   18     10
```

```
19     10 process turnstile2;
20     11
21     11 var
22     11    loop: integer;
23     11 begin
24     11    for loop := 1 to 20 do
25     15       count := count + 1
26     18 end;  (* turnstile2 *)
27     21
28     21
29     21
30     21

31     21 begin
32     22    count := 0;
33     25    cobegin
34     26       turnstile1;
35     30       turnstile2
36     30    coend;
37     35    writeln('Total admitted: ',count)
38     39 end.
```

2.1. Comments on the listing

It will be seen that the compiler has pre-pended two integers to each line of the original source code. The first is clearly a line number. The second is the instruction counter for the generated code, and it represents the count as it was when the compiler began to process the line concerned. This information is useful because the interpreters quote the instruction counter when they encounter a run-time error, so that the approximate location of the problem can be determined.

APPENDIX B - COMPILE-TIME ERRORS

1. Types of error

Error-messages generated by the Pascal-FC system fall into two categories:

(1) compile-time errors;

(2) run-time errors.

Run-time errors are considered in Appendix C.

The compile-time errors may be further divided into the following sub-categories:

(a) fatal errors, usually due to compiler internal table overflow;

(b) violations of the syntax of Pascal-FC;

(c) use of optional Pascal-FC facilities not supported by this implementation.

2. Fatal errors

Most of these errors are caused by overflow in one of the compiler's internal tables. The two exceptions are:

- **program incomplete.** This message is generated when the compiler unexpectedly reaches the end of the program source file.
- **input line too long.** An internal line buffer limits the length of lines in the source program. The limit was specified in Table 1.1.

The remaining fatal errors, which are all due to table overflow, all produce messages which name the table involved. The following notes provide additional information to assist in the interpretation of these messages.

1 **identifier.** The symbol table has overflowed. This table is used to record all identifiers used in a program.

2 **blocks.** Every subprogram, monitor, resource, record type and entry declared in a program is recorded in the block table.

3 **strings.** All string literals used in the source program are recorded in this table.

4 **arrays.** Each new array type declared is recorded in this table. This includes "anonymous" array types introduced in `var` declarations.

5 **levels.** The degree of nesting of blocks or of record types is excessive.

6 **code.** The source program has generated too much code.

7 **channels.** This is analogous to the case for arrays. Each new channel type declared in the program is recorded in the channels table.

8 **select.** There are too many alternatives in a `select` statement. Note that a `replicate` alternative counts as a single alternative.

9 **monprocs.** There are too many exported procedures in a monitor or resource.

10 **reals.** Every unique real literal used in a program is entered into this table.

11 **`interrupts`**. Every semaphore, channel or entry declared with a mapping indicator is entered into this table.

12. **enum type.** Each new enumeration type declared requires an entry in this table. This includes "anonymous" types introduced in **`var`** declarations.

13 **case**. There are too many alternatives in a **`case`** statement.

14 **monitors.** Too many monitors or resources have been declared in the program.

The values of the constants governing the sizes of the above tables will be found in Table 1.1.

3. Non-fatal Errors

3.1. List of messages

Some of the error messages may require the brief explanations given below. Others are self-explanatory, and have been simply listed for completeness.

E0 **Undeclared identifier.** A reference has been made to an identifier which has not been declared, or is not in scope.

E1 **Identifier duplicated.** An attempt has been made to declare an identifier which has already been declared in the current scope.

E2 **Identifier expected.** This error may be generated in numerous circumstances, such as in constant, type or variable declarations and in the formal parameter lists of subprograms.

E3 **Type error.** Generated in many circumstances where operands have invalid types.

E4 **"(" expected.**

E5 **")" expected.**

E6 **"[" expected.**

E7 **"]" expected.**

E8 **":" expected.**

E9 **";" expected.**

E10 **"." expected.**

E11 **"=" expected.**

E12 **":=" expected.**

E13 **"program" expected.** This is generated if the first line of the source file does not begin with "program". (Comments excepted).

E14 **"of" expected.**

E15 **"then" expected.**

E16 **"until" or "forever" expected.**

E17 **"do" expected.**

E18 **"to" expected.**

E19 **"begin" expected.**

E20 **"end" expected.**

E21 **"select" expected.**

E22 **"export" expected.** The first item in a monitor or resource declaration must be an export list.

E23 **"replicate" expected.** When using the replicator in the select statement for rendezvous by channel, the "replicate" has been omitted ("do" may have been erroneously used instead).

E24 **Error in parameter list.** A formal parameter list does not begin with an identifier or with **`var`**, or the actual parameters in a subprogram call or entry call do not match the associated formals.

E25 **Must be var parameter.** Semaphore, condition and channel objects may be passed as parameters to subprograms (including processes), but they must be formally declared as **`var`** parameters. If they were permitted to be passed as value parameters, then any operations carried out within the subprogram would access and modify only a local copy, which would defeat their purpose for inter-process communication.

E26 **Parameter list does not match previous declaration.** In an **`accept`** statement, the formal parameters of the entry do not match exactly those given in the declaration of that entry.

E27 **Illegal character.**

E28 **Unexpected symbol.** This error can be generated in many circumstances when the next lexical symbol in the source program is not what the compiler expects.

E29 **String expected.**

E30 **Level error.** A language feature has been used at a static level which is inappropriate for that feature. For example, an inter-process communication primitive has been declared elsewhere than in the declaration part of the main program (Level 1), or an entry call has been made in the main program statement part.

E31 **Number error.** An error has been detected in an integer or real literal. For example, an integer literal is too large for the implementation.

E32 **Assignment not permitted.** The variable on the left of an assignment operator is an inter-process communication type, or is a structured object having such a type as a component.

E33 **exported monitor/resource procedure(s) not declared.** One or more identifiers appeared in the **`export`** list of a monitor or resource for which no corresponding procedure declaration was found. The message will appear following the final **`end`** of the offending monitor or resource.

E34 **Must not be var parameter.** The control variable of a `for` statement or `replicate` alternative may be a variable declared within the same block, a variable declared in an enclosing block, or a *value* parameter of the subprogram containing the `for` or `replicate`. It must not, however, be a variable parameter of that subprogram.

E35 **Malformed entry call.** The compiler has interpreted a statement as an entry call, but its form is incorrect. As well as errors in statements which the programmer actually intended as entry calls, this message will be generated if an attempt is made to activate a process anywhere outside the concurrent statement of the main program.

E36 **Not allowed in a process.** A language feature has been used that should not appear in a process declaration (for example, a nested process declaration).

E37 **This type must not be mapped.** It is illegal to use a mapping indicator with any condition object, or any structured object which contains a semaphore object or a channel object.

E38 **"Timeout", "terminate" and "else" mutually exclusive.** More than one of these mutually exclusive alternatives has been used in a `select` statement.

E39 **Multiple cobegins.**

E40 **"Forward" declarations(s) not resolved.** The declaration part of a block contains a `forward` declaration of a procedure or function for which no full declaration has been given. This message is signalled at the end of the declaration part concerned.

E41 **"Provides" declaration(s) not resolved.** The `provides` construction has been used to predeclare the entries (and possibly the parameters) of a process or process type, but a full declaration of the process or process type has not been given.

E42 **Variable expected.**

E43 **Missing entry or entries declared in "provides".** This error can be generated during the parsing of the full declaration of a process which has been pre-declared in a `provides` construction.

E44 **Case label duplicated.** In a `case` statement, two or more alternatives have the same case label.

E45 **Processes not allowed in record fields.** A process, or array of processes, has been declared as a field of a record.

E46 **Invalid set literal.** A bitset literal is a list of integer expressions enclosed in square brackets. A set literal has been used that contains a value which is not an integer expression.

E47 **Variable is an array, not a process**. When an array of processes has been declared, each component process must be separately activated in the concurrent statement: "whole array" activation is not permissible. This error will result if insufficient subscripts are given when attempting to activate a process declared as an array element.

E48 **Error in array subscript declaration.** In the declaration of an array, the type and range of each subscript is indicated by a constructions of the form *lower..higher*. If the ordinal value of *lower* is in fact greater than that of *higher*, this error is generated. In addition, there is a limit set on the ordinal value of both of these values. Exceeding these limits also generates this error.

E49 **Constant expected.**

E50 **No corresponding "provides" declaration.** In parsing the full declaration of a process which was pre-declared using the `provides` construction, an entry declaration has been found that was not included in the `provides` declaration.

E51 **Does not match "provides" declaration.** This error is associated with declarations of entries mapped to interrupt sources. Either the entry concerned was mapped in the `provides` construction but not in the full declaration (or vice versa) or the integer identifying the interrupt source is not the same in the two cases.

E52 **Illegally nested accept.** An `accept` statement for a particular entry is nested inside an `accept` statement for the same entry.

E53 **`accept not allowed in subprogram.`** An `accept` statement is only allowed in the statement part of a process.

E54 **`not allowed in a guarded procedure.`** An attempt has been made to nest a guarded procedure declaration inside a guarded procedure declaration.

E55 **`only allowed in a guarded procedure.`** A `requeue` statement can only be placed in the statement part of a guarded procedure.

E56 **`destination must be guarded procedure.`** The destination of a `requeue` statement must be a guarded procedure in the current or some other resource.

E57 **`only allowed in a resource.`** Guarded procedures may only be declared in resources (not monitors).

E58 **`call not allowed within a resource`**. An ordinary procedure call to a guarded procedure of a resource is only allowed from outside a resource.

3.2. Listing file diagnostics for these errors

When any of the above errors are detected in a program, the error diagnostics are sent to the listing file produced by the compiler, and a rough indication of the location of the errors is also given in the listing of the source code. In general, the position at which the compiler finally concludes that an error has occurred is what is indicated, and this is sometimes a little way beyond the actual site of the error (possibly on a subsequent line). The following example illustrates the form of the error diagnostics. Note, for example, that error 9 has not been detected immediately. This program has also generated a fatal error due to a typographical error in the final `end`.

```
- Pascal-FC for Unix systems -
- Compiler Version 5.1 -

G L Davies  &  A Burns, University of Bradford

Compiler listing for file errorprone.pfc



    1     0 rogram errorprone;
***********      ^13---------
    2     0
    3     0 const
    4     0    1 = 1;
***********    ^ 2--
    5     0
    6     0 var
    7     0    x: reel;
***********          ^ 0
    8     0    i,@j: integer
***********      ^27
    9     0    ch: char;
***********     ^ 9
   10     0 @
*********** ^27
   11     0
   12     0 begin
   13     0    ch := 1;
***********           ^ 3
   14     2    i := '1' + '2';
***********                  ^ 3
   15     5    j := 3
   16     7 ebd.
***********   ^28
   17     8
```

```
 Error diagnostics

E0 -  undeclared identifier
E2 -  identifier expected
E3 -  type error
E9 -  ';' expected
E13 -  'program' expected
E27 -  illegal character
E28 -  unexpected symbol
FATAL ERROR - program incomplete
```

Following some kinds of error, the compiler is unable to return immediately to the normal processing of input. In these cases, the input is ignored until normal processing has been resumed. Sections of input ignored in this way are underlined, as in Line 1 of the above example.

4. Use of unsupported options

Though this implementation does not support mapping indicators and offset indicators, the compiler will still check the validity of the syntax of such features in the source program and report on any errors. In addition, a message will be given to warn that these facilities are not supported, and the program will not be executable.

APPENDIX C - RUN-TIME ERRORS

In all cases, the generation of a run-time error results in the immediate termination of the program. In most cases, the approximate position where the error occurred is indicated by a message which indicates the value of the *pc* (the generated code counter) at the time the error occurred. This counter points to the next instruction which would have been executed had no error occurred.

1. Deadlock

The scheduler is unable to find an "executable" process, and at least one of the processes in the program is "suspended". Chapter 3 of the Pascal-FC Language Reference Manual discusses deadlock.

2. Channel error

Two or more receivers (or two or more senders) are attempting to rendezvous on the same channel. Channels are for point-to-point communication between *one* receiver and *one* sender.

3. Closed guards

In the execution of a **`select`** statement, all guards have evaluated to `false`, and there is no **`else`** part.

4. Attempt to call entry of non-existent/terminated process

A call has been made on an entry of a process that was never activated, or is now "terminated".

5. Multiple activation of a process

An attempt has been made to activate several instances of a process in the concurrent statement.

6. Division by zero

This can result from real division or from integer division (using the **`div`** and **`mod`** operators).

7. Invalid index

An array subscript is out of range.

8. Illegal character

The argument given to the `chr` function is outside the range 0..127.

9. Storage overflow

Memory space allocated for the main program or for a process has been exhausted. This is likely to be caused by large data structures or by deeply nested subprogram calls.

10. Reading past end of file

An attempt has been made to `read` when there are no more characters in the input file. This message is relevant when input redirection is being used.

11. More than 100 processes

There is no specific limit on the number of processes that may be *declared*, but this error will be generated if an attempt is made in a program to *activate* more than 100.

12. Statement limit of 200000 reached (possible livelock)

The run-time system will abort the program if the statement limit is reached. A common cause is livelock, but it may simply be that the program's execution is too long.

13. Label of ... not found in case

The selector expression in a **`case`** statement has a value for which there is no corresponding alternative.

14. Ordinal value out of range

This can be generated during the execution of the `pred` and `succ` operators, or during the semaphore `initial` procedure if a negative argument is given.

15. Error in numeric input

When executing a `read`, an unexpected character has been found, or a numeric value has been read which is out of range for this implementation, or which uses a number base other than binary, octal, hexadecimal or decimal.

16. Bitset value out of bounds

There are three circumstances in which this error can occur:

(1) when using the `bits` type-transfer function, the integer argument supplied cannot be converted into a bitset without truncation;

(2) when using the **`in`** operator, the left operand is not in the range 0..7;

(3) in a set literal, the integer expressions do not all lie in the range 0..7.

17. Arithmetic overflow

This error can result both from integer and real arithmetic operations when the result is too large (or too negative) to be represented. It is also generated in two cases where an inappropriate argument has been passed to a standard function:

(1) a negative or zero argument to `ln`;

(2) a negative argument to `sqrt`.

18. Attempt to initialise semaphore from process

The main program thread (including procedures and functions called by the main program, to whatever depth) is the only context in which the `initial` procedure can be called.

APPENDIX D - DISCLAIMER AND REPORTING OF FAULTS

Though care has been taken to avoid faults in the Pascal-FC implementations produced at Bradford, the software is provided without any form of warranty, and the authors cannot accept any liability arising from its use in any circumstances.

We wish to eliminate any hitherto undetected faults in any of our implementations. Hence, we invite users who discover faults to report them to: Dr G.L.Davies, Department of Computing, University of Bradford, Bradford, West Yorkshire, UK, BD7 1DP (email: G.L.Davies@bradford.ac.uk). Please supply the following information:

1. A listing of the Pascal-FC source program;
2. any resulting *listfile*;
3. any resulting *pmdfile*;
4. output results (where appropriate);
5. a short description of the problem;
6. an address (preferably electronic mail) where we can contact you.

CONTENTS