TABLE OF CONTENTS

COPYRIGHT

NOTICE

Gambit Games reserves the right to make improvements in the product described in this manual at any time and without notice.

G-Pascal is designed, written, manufactured and supported in Australia by Gammon & Gobbett Computer Services Proprietary Limited trading as Gambit Games.

Gammon & Gobbett Computer Services Proprietary Limited is a company incorporated in the State of Victoria.

This Manual was typeset by Hughes Phototype, Cremorne, NSW, and printed by Noosa Graphica, Noosaville, Qld.

DISCLAIMER

G-Pascal is sold or licensed "as is". The entire risk as to its quality and performance is with the buyer. Should G-Pascal prove defective following its purchase the buyer (and not Gambit Games, its distributor, or its retailer) assumes the entire cost of any necessary correction and any incidental or consequential damages. Gambit Games believe G-Pascal and this Manual are accurate and reliable and much care has been taken in their preparation, however Gambit Games make no warranties, either express or implied, with respect to this manual or with respect to the G-Pascal compiler, its quality, performance, merchantability, or fitness for any particular purpose.

REPLACEMENT POLICY

Gambit Games warrants to the original purchaser only the medium on which G-Pascal is recorded to be free from defects in materials or workmanship under normal use and service for a period of ninety (90) days from the date of purchase. If during this period a defect on the medium should occur, the medium may be returned to Gambit Games or to an authorised Gambit Games dealer, and Gambit Games will replace the medium without charge to you. Your sole and exclusive remedy in the event of a defect is expressly limited to replacement of the medium as provided above.

To provide proof that you are the original purchaser please complete and mail the enclosed Owner Warranty Card to Gambit Games.

If the failure of the medium, in the judgement of Gambit Games, resulted from accident, abuse or misapplication of the medium, then Gambit Games shall have no responsibility to replace the medium under the terms of this warranty.

POSTAL ADDRESS

Please address correspondence to:
Gambit Games,
P.O. Box 124,
Ivanhoe,
Victoria 3079.
Australia.

INTRODUCTION

Congratulations on purchasing G-Pascal! We are sure that you will find it very useful and easy to use. G-Pascal on the Commodore 64 contains very advanced features making it an excellent development tool for this computer, namely:

- High speed compiler (6,000 lines per minute) which implements a comprehensive subset of standard Pascal.
- Built-in powerful Text Editor which includes global Find and Replace capability.
- *Extensions provide extensive support for the Commodore 64's graphics, music and sound effects, time-of-day clock, interval timer, cursor control and colour control.* There are in fact 76 separate functions and actions built-in as extensions for the Commodore 64. Sprite handling is particulary well catered for, with commands to automatically move sprites around the screen with animation if desired, and to automatically stop sprites if they collide.
- Complete arcade-style games can be written without using a single PEEK or POKE.
- Error messages are in plain English with an arrow to the point of error.
- As the entire system is memory-resident, editing, compiling and testing is very fast and easy.
- Pascal programs may readily be stored on disk or cassette – programs are stored in a 'compressed' format, thereby speeding up loading and saving times, and allowing a larger program to fit into memory.
- Support for machine-code (assembler) subroutines if desired.
- Debugging aids, such as Trace and Debug modes, which can be invoked at any time from the keyboard.
- Compiler supports INTEGER and CHAR data types, and single- dimension arrays. Integers range from –8388608 to +8388607.
- Compiler supports the standard Pascal constructs: CONST, VAR, PROCEDURE, FUNCTION, WHILE, DO, REPEAT, FOR, IF and CASE.
- Arithmetic expressions may contain the relational operators as well as +, –, /, *, MOD, AND, OR, XOR, SHL, SHR and ABS.
- Compiler supports independent modules (Procedures that are compiled independently and located elsewhere in memory).
- Compiler produces relocatable P-codes – the object code produced by the compiler may be run at any memory address without change.

Run-time package

If you are planning to develop commercial programs or games in G- Pascal please enquire about our interpreter-only Run-time package. The Run-time system consists of the interpreter as a stand-alone program which is 6K long. Using the Run-time system simply consists of loading the P-codes produced by this compiler and then running the interpreter.

This Manual

This Manual has been designed with a number of purposes in mind:

- To introduce the Pascal language.
- To fully describe G-Pascal's capabilities and be a 'reference manual' for the experienced programmer.
- To describe the G-Pascal support system, such as the Editor, File system, error messages and so on, so that the G-Pascal system is easy to understand and use.

It falls beyond the scope of this Manual to provide a really comprehensive guide to the Pascal language. There are many good books on the Pascal language in bookshops and libraries, containing between them thousands of pages of information and examples about Pascal, good programming practices, games and serious applications. You are strongly recommended to find a Pascal book which covers the type of programs that you are interested in (games, mathematics, business, adventure and so on).

G-Pascal and this Manual are somewhat oriented towards arcade- style games. The reason for this is that the Commodore 64 with its sprites and 3-voice synthesizer is an excellent and low-cost method of easily writing and experimenting with arcade-style games. Consequently the discussions and examples (particularly about sprites) seem the most meaningful in this context.

However there is no reason why G-Pascal cannot be used for 'text' oriented programs, such as adventure games or more serious applications, for example: small-scale business software, mathematics or probability experiments, and educational software.

Suggestions

If you have just purchased G-Pascal and don't know where to start, these suggestions may be helpful:

- First, load G-Pascal into your Commodore 64 so that you can try out the examples for yourself, and confirm that G-Pascal does what this Manual says it will.
- Turn to 'Getting Started With G-Pascal' (over the page) and follow the step-by-step instructions to enter, compile and run your first program.
- While your first program is still in memory start experimenting! Try different Editor commands – try saving the program to disk or cassette and re-loading it – try tracing the program – try changing the program.
- G-Pascal is designed to be 'fail-safe'. Built-in checks make it very hard to do something 'wrong'. For example, it will not allow a program to be run before it has been compiled. The very worst that can happen is that a program wipes out itself or G-Pascal (possibly making it necessary to turn off the power and re-load G-Pascal) however this is extremely rare.
- If you are a beginner to Pascal, or want to know in detail about what G-Pascal will do, read the section 'Beginner's Guide to Pascal'.
- Skim through the Manual, becoming familiar with what G- Pascal can do. Try out the examples – most of them are quite small and only take a couple of minutes to type in.
- Experiment in more detail with the aspects of G-Pascal (and the Commodore 64) that you are interested in (e.g. graphics, sound effects, games, etc.)

HOW TO LOAD AND RUN G-PASCAL

1. Load and run G-Pascal as directed by the instructions that came with your disk, cassette or cartridge.
2. Your G-Pascal is now running! You will see:

```
G-Pascal compiler Version 3.0 Ser 4321
Written by Nick Gammon and Sue Gobbett
Copyright 1982 Gambit Games
P.O. Box 124 Ivanhoe 3079 Vic Australia

<E>dit, <C>ompile, <D>ebug, <F>iles,
<R>un, <S>yntax, <T>race, <Q>uit ?
```

Now see the section 'Getting started with G-Pascal' which is a step-by-step guide to entering your first program.

SPECIAL KEYBOARD FUNCTIONS

The following keys have special meaning to G-Pascal.

Space bar

Pressing 'space' will freeze a display on the screen. Press any key (except RUN/STOP) and the display will continue. RUN/STOP will abort the display.

RUN/STOP

This will abort a display, or if a program is running, will abort the program with a message showing which P-code was currently being executed.

RUN/STOP/RESTORE

Holding down the RUN/STOP key and then pressing RESTORE will do a 'warm boot' aborting whatever G-Pascal is doing, clearing the screen, and returning to the Main Menu. Any program in memory is not lost, however. Normally just pressing RUN/STOP (without RESTORE) will be sufficient to stop a program running, however press RESTORE as well if nothing seems to be happening.

COMMODORE/T

If a program is running will cause a Trace to start. (See 'Debugging your program' for more details). *To enter COMMODORE/T hold down the key with the Commodore logo (bottom left hand corner of the keyboard) and then press the 'T' key.*

COMMODORE/D

If a program is running will cause Debug mode to start. (See 'Debugging your program' for more details). *To enter COMMODORE/D hold down the key with the Commodore logo (bottom left hand corner of the keyboard) and then press the 'D' key.*

COMMODORE/N

If a program is running will stop a Trace or Debug. (N = Normal). If the program is not Tracing or Debugging pressing COMMODORE/N will have no effect. *To enter COMMODORE/N hold down the key with the Commodore logo (bottom left hand corner of the keyboard) and then press the 'N' key.*

Arrow keys

The arrow keys (and INST/DEL key) function the same as in Basic – useful when in the Editor or when typing in a file name to Load or Save.

GETTING STARTED WITH G-PASCAL

This section provides a step-by-step introduction to using the G- Pascal system. We will key in, compile and run a simple program. Please work through this example using your Commodore 64 – it will give you familiarity with the various phases of the system (editing, compiling and running).

To distinguish between what you enter and what G-Pascal responds, all input to be entered by you is in ***bold italics.*** All Editor commands are terminated by pressing the RETURN key.

After loading G-Pascal you will see the Main Menu – enter 'E' to invoke the Editor ...

```
(E)dit, (C)ompile, (D)ebug, (F)iles,
(R)un, (S)yntax, (T)race, (Q)uit ? E
```

The Editor prompts with a colon (:). Type L (for List) to confirm that there is no program currently in memory ...

```
:L
```

Now enter your program by placing the Editor in input mode. To do this, type I (for Input) ...

```
:I
```

The Editor will now prompt you with a '1' – indicating that you are about to enter line number 1 of your program. Type in the program as shown, pressing RETURN at the end of each line. If you make a typing mistake while entering a line, you can use the INST/DEL or cursor movement keys (arrow keys) to correct it ...

```
1 (* program to display the squares
2 of numbers from 1 to 10 *)
3 const limit = 10;
4 var k : integer;
5 begin
6 for k := 1 to limit do
7 writeln ("square of ",k," is ",k * k)
8 end.
9
```

Just press RETURN on its own here and the Editor will leave Input mode, and display a colon again, indicating it is ready for another command.

```
:
```

Now list your program by typing L again. You should now see a listing of your program which looks identical to the program above ...

```
:L
```

If you notice a mistake at this stage the easiest way to correct it is to use the Editor's Modify command. For example, if you misspelt 'begin' on line 5 as 'began', then you would type in 'M5' to modify line 5. The Editor will echo the current contents of line 5, and then enter Input mode to allow you to type one or more lines to replace it. The whole operation would look like this ...

```
:M 5
5 began
5 begin
6
```

Rather than retyping the line you may wish to just move the cursor over the line in error (using the arrow keys), correct the line and then press RETURN.

Just press RETURN here to leave Input mode again. You can now attempt to compile it by entering C (for Compile). Compiling the program converts it to object code (P-codes) ready for Running.

If you have made an error in typing in your program you will get an error message, and be returned to the Editor, with the colon (:) displayed again. If this happens, Modify the offending line and try again.

If you have not made any errors the whole process will look like this ...

```
:C

G-Pascal compiler Version 3.0 Ser# 4321

Written by Nick Gammon and Sue Gobbett
Copyright 1983 Gambit Games
P.O. Box 124 Ivanhoe 3079 Vic Australia

P-codes ended at 412E
Symbol table ended at C026

(C)ompile finished: no Errors

(E)dit, (C)ompile, (D)ebug, (F)iles,
(R)un, (S)yntax, (T)race, (Q)uit ?
```

At this stage the Main Menu will be displayed again – now enter R (for Run) and you will see the results being displayed on the screen ...

```
(E)dit, (C)ompile, (D)ebug, (F)iles,
(R)un, (S)yntax, (T)race, (Q)uit ? R
Running
square of  1 is   1
square of  2 is   4
square of  3 is   9
square of  4 is  16
square of  5 is  25
square of  6 is  36
square of  7 is  49
square of  8 is  64
square of  9 is  81
square of 10 is 100

run finished – press a key ...
```

G-Pascal now displays its 'end of run' message, as above. This is to give you a chance to read the results. When you press a key (such as RETURN), the screen will clear and you will see the Main Menu again ...

```
(E)dit, (C)ompile, (D)ebug, (F)iles,
(R)un, (S)yntax, (T)race, (Q)uit ?
```

You can now enter the Editor again and try something else! You may want to experiment with minor changes to this demonstration program to get used to the Editor and the Compiler. This would be a good time to read the part of this Manual pertaining to the Editor and experiment with the other Editor commands, such as Find and Replace. When you are confident about the Editor try some of the other examples in the book, such as those demonstrating the various graphics and sound effects capabilities of G-Pascal. If you want to delete the program currently in memory so that you can 'start from scratch', just enter: 'D 1-9999' as in the following illustration ...

```
(E)dit, (C)ompile, (D)ebug, (F)iles,
(R)un, (S)yntax, (T)race, (Q)uit ?E

:D 1-9999

Do you want to delete 8 lines ? Y/N Y

8 deleted

:L

:
```

For a summary of Editor commands, just enter 'H' (Help).

HOW TO USE G-PASCAL

After loading and running G-Pascal or after a compile or run, you will see the 'Main Menu' :

<E>dit, <C>ompile, <D>ebug, <F>iles,
<R>un, <S>yntax, <T>race, <Q>uit ?

It is called a menu because you have various choices to make, depending on what you want.

To choose one, press the letter corresponding to your choice (the letter in brackets). For example press C for Compile. Do not press the RETURN key as well. If you make an incorrect choice the Commodore 64 will re-display the Main Menu.

The choices are :

Edit

Enters the Editor to allow you to type in or change a Pascal program. When the Editor is active it will display a colon (:) to let you know it is awaiting an Editor command. See 'Using the Editor' for more details.

Compile

Compiles a program (that is, converts it to P-codes) that you have loaded or edited. If the compile has no errors you may then type R (for Run) if you want to run your program. A program must be compiled before it can be run. See 'Compiling your program' for more details.

Syntax

This does a 'syntax check' of your program, letting you know if it has any errors. It does not generate P-codes however so you must do a compile as well before running the program. See 'Compiling your program' for more details.

Run

This will run your program. It must have been successfully compiled first or the message 'No valid compile done before' will be displayed. G-Pascal will display:

Running

and then commence running your program.

Debug and Trace

These are a variation of the 'Run' command which, in addition to running your program, display debugging information as your program runs. You will not normally choose these options as the information they display clutters up the screen somewhat and slows down execution of the program considerably. See 'Debugging your program' for more details.

Files

Enters the 'Files Menu' to allow you to load or save programs, turn on or off your printer, and use the Commodore 64 DOS to delete files, etc. See 'File handling' for more details.

Quit

Leaves G-Pascal, but first checks that you wanted to quit by asking:

Quit ? Y/N

If you do not type 'Y' the quit has no effect. Do not Quit unless you have finished with G-Pascal for now.

USING THE EDITOR

The Editor displays a colon (:) when it is ready for a command, so if you see a : you know you are in the Editor. To leave the Editor enter: Q (RETURN) and you will return to the Main Menu, or QF (RETURN) and you will go straight to the Files Menu.

Line numbers

The Editor is line-number oriented. This means that all Editor commands refer to the line-numbers of each line in your program. The line numbers are allocated automatically by the Editor, starting at 1 and going up by 1 each line. The line number allocation is 'dynamic', that is, if you insert a line between what is currently line 6 and line 7 then the new line becomes line 7 and the line that used to be line 7 becomes line 8, etc.

Commands

All Editor commands consist of *one* letter followed by none, one or possibly two line numbers. For example:

```
L 10,20
```

would list lines 10 to 20. In some commands (Find and Replace), the line number range is also followed by a 'string' enclosed within delimiters. If you are specifying a line-number range, there should be either no spaces, or one space between the command and the line numbers.

If you are specifying two line numbers (as in the above example) you can use any non-numeric 'delimiter' between the line numbers (except '.' which is used by Find and Replace to delimit strings). Therefore use whatever you feel comfortable with. For example, all the following are equivalent:

```
L 15-30
L 15 30
L 15,30
```

The commands are :

```
<C>ompile
<D>elete Line Number Range
<F>ind Line Number Range . string .
<I>nsert Line Number
<L>ist Line Number Range
<M>odify Line Number Range
<Q>uit
<R>eplace Line Number Range . old . new .
<S>yntax
```

(The above command summary appears if you type 'HELP' when in the Editor).
The commands are explained in detail on the following pages.

EDITOR COMMANDS

DELETE

Deletes one or more lines. Any lines following those deleted are renumbered.

To delete one line:

D *line-number*

e.g.

D 5

would delete line 5.

To delete a range of lines:

D *first-line, last-line*

e.g.

D 10,20

would delete lines 10 to 20.

If you attempt to delete 5 or more lines the Editor will ask you (for example):

Do you want to delete 200 lines ? Y/N

This is in case you meant to delete 20 lines but accidentally tried to delete 200 lines. If you want the delete to go ahead press 'Y' otherwise press any other key. You do not need to press (RETURN) as well.

INSERT

Inserts new lines into your program. To insert, enter:

I *line-number*

where 'line-number' is the line that you want to insert *after*. To stop inserting just type (RETURN) when the cursor is over a blank line (or a line containing only the line number). Usually this means just press RETURN before typing in anything. To insert a blank line enter SHIFT/SPACE and then press RETURN.

For example:

I 10

inserts after line 10 (starting at line 11).

The Editor reminds you what line you are currently inserting by displaying its line number at the start of the line. To put a line at the very start of the program (line 1) enter:

I

To put a line at the end of the program enter:

I 9999

LIST

Lists your program.

To list the whole program: L

To list one line: L *line-number*

To list a range of lines:

L *first-line,last-line* e.g.

L 10

will list line number 10.

L 40, 60

will list lines 40 to 60.

You can temporarily 'freeze' your list by pressing the space bar. You can stop the listing altogether by pressing the RUN/STOP key.

To list without line numbers:

There is a variation of the List command called the 'No-line- number List' (N for short). It works the same as List except that line numbers are not displayed on the left. Its intended purpose is for listing the file when the line numbers would be a nuisance – for example if you use the editor for non-Pascal uses such as producing a letter or a name and address list. The 'N' command does not appear on the command summary.

e.g.
N 100-500
would list lines 100 to 500 without line numbers.

MODIFY

Allows you to change one or more lines in your program.

To change one line:

M *line-number*

To change a range of lines:

M *first-line, last-line*

Modify works by:

a) listing the line or lines requested.
b) deleting the lines requested.
c) entering Insert mode so that you can replace them.

Like the Delete command, Modify will warn you if you are about to change more than 4 lines.

The intention is that for making minor changes to small numbers of lines you use the 'cursor control' keys to move the cursor over the erroneous lines, making corrections where necessary. Once a line is corrected just press RETURN to copy the corrected line back into the program. The cursor does not have to be at the end of the line – anywhere on the line will do.

COPYING AND MOVING LINES

The Insert command can be used to copy or move text from one part of the program to another (in groups of up to 20 lines). The method of doing this is:

a) List the lines to be copied/moved.
b) Enter 'Insert' mode at the point where the lines are to be copied or moved to.
c) Move the cursor to the start of the lines listed (in (a) above) using the upwards cursor control key.
d) Press RETURN to copy that line into the new spot. If more than one line is to be copied keep pressing RETURN until all the desired lines are copied. Then leave Insert mode by positioning the cursor over a blank line and pressing RETURN (an easy way to do this is to press SHIFT/CLEAR/HOME and then RETURN).
e) Delete them from the old place in the program if desired. (Warning - they may have different line numbers now if you inserted before them.)

To move or copy larger blocks of code (hundreds of lines) you can use the 'append' command in the file menu – just save to disk or cassette a temporary file containing the code to be moved and then append it back into the right spot.

FIND

The Find command is a powerful method of quickly locating a string of text within your program. By 'string' we just mean a sequence of characters.

For example – to find the start of every procedure in your program:

```
F.procedure.
```

As the editor finds each line containing the word 'procedure' (in this example) it displays that line with its line number. When it has finished it displays the total number found.

The Find command can also be used on a range of lines (or a single line) by specifying a line range in the usual way –

F *first-line, last-line . string . options*

e.g.

```
F 100-200.begin.
```

The string delimiters *must* be the '.' symbol, they must *both* be present, and directly follow the line number range (if present) or the letter 'F' (if not). Pressing RUN/STOP will abort the Find.

Options on the Find

The second delimiter may optionally be followed by up to three parameters – each represented by a single letter. They may appear in any order or not at all. The options are:

T – Translate – this will temporarily translate any lower-case letters in the program to upper case before comparing them to the specified string. If the 'T' option is *not* used then the Find will differentiate between 'FRED' and 'fred', for example. Since the compiler considers upper and lower case identical (except inside string literals) then it may be wise to use the 'T' option when checking for how often words occur. If 'T' is used then the string specified in the Find *must* itself be entered in lower- case.

e.g.

```
F.fred.T
```

Q – Quiet – this will not display lines which contain the specified string – only the count at the end of the Find will be displayed. This is useful for just counting the number of times (for example) the word 'BEGIN' occurs without actually listing all the lines which contain it.

e.g.

```
F.begin.q
```

G – Global – this will search each line in the program for *every* occurrence of the specified string, not just the first. The only difference this will make to the Find is that if a line contains more than one occurrence of the string then the count at the end of the Find will be different.

e.g.

```
F.begin.gtq
```

REPLACE

The Replace command is a powerful method of quickly replacing a string of text within your program with another one. By 'string' we just mean a sequence of characters.

For example – to replace the first occurrence of the word 'enemies' on each line with 'klingons' just say:

```
R.enemies.klingons.
```

As the editor finds each line containing the word 'enemies' (in this example) it replaces the *first* occurrence on that line with the word 'klingons' and then displays the line with its line number. When it has finished it displays the total number replaced.

The Replace command can also be used on a range of lines (or a single line) by specifying a line range in the usual way –

R *first-line, last-line. old string . new string . options*

e.g.

```
R 100-200.fred.nurk.
```

The string delimiters *must* be the '.' symbol, all three *must* be present, and directly follow the line number range (if present) or the letter 'R' (if not). Pressing RUN/STOP will abort the Replace.

Options on the Replace

The third delimiter may optionally be followed by up to three parameters – each represented by a single letter. They may appear in any order or not at all. The options are:

T – Translate – this will temporarily translate any lower-case letters in the program to upper case before comparing them to the target string. If the 'T' option is *not* used then the Replace will differentiate between 'FRED' and 'fred', for example. See the discussion under the 'Find' command for more details.

Q – Quiet – this will not display lines which have been replaced - only the count at the end of the Replace will be displayed. This is useful if you know there will be a lot replaced – and you are sure that you are replacing the right things!

G – Global – this will replace *every* occurrence of the string on each nominated line – not just the first – use with discretion. *Warning:* if you use the 'Global' option indiscriminately and make a program line longer than 88 characters (the maximum the system can handle) then the Editor will not function correctly. For space reasons there is no check built in to prevent this happening, so make sure that in the process of replacing that program lines are kept to a reasonable length.

COMPILE

This starts a compile of your program. It is the same as entering 'C' from the 'Main Menu'. See 'Compiling your program' for more details.

SYNTAX

This does a 'syntax check' of your program (to let you know if it has errors). It is the same as entering 'S' from the 'Main Menu'. See 'Compiling your program' for more details.

QUIT

Typing 'Q' leaves the Editor (quits) returning you to the 'Main Menu'. Typing 'QF' takes you directly to the 'Files' menu.

BEGINNER'S GUIDE TO PASCAL

This guide is not intended to be a comprehensive Pascal tutorial, however it will introduce you to G-Pascal's basic concepts. Try out the examples yourself to become familiar with the language structure and what the statements do.

General format

Spaces or new lines may freely be used within your program (except in the middle of words or symbols) to make the program more readable. Upper or lower case characters may be used interchangeably as the compiler converts everything to upper case internally (except string constants, e.g. "fred").

Comments

Comments may be freely interspersed within your program – they must be enclosed within the (* and *) symbols. Comments may appear anywhere (except within the middle of a word or symbol).

Reserved words

Reserved words are those that have a specific meaning to the compiler and must be used in the correct context. They are shown in this section in upper case in order to distinguish them from other words. However they may be entered in lower case in your programs if you wish. In fact, since all reserved words are 'tokenised' by the compiler they will always be displayed in lower case, regardless of how they are entered.

User-defined words

All words other than reserved words are chosen by the programmer (yourself). They may be any length (all characters are significant), must start with a letter, and after that consist of letters, numbers or the underscore character. (The underscore is shown in this Manual as _ however on the Commodore 64 it looks like a 'left-arrow', and is entered by the key on the top left-hand side of the keyboard.) All user-defined words must be declared before they are used. 'Declaring' a word means telling the compiler to what use you are putting the word. You do this by using the word in a CONST, VAR, PROCEDURE or FUNCTION declaration.

GENERIC WORDS

In this section of the Manual 'generic' words are entered in *italics*. For example, the definition of an IF statement is:

IF *expression* THEN *statement* ELSE *statement*

In this case the words *expression* and *statement* are neither reserved words or user-defined words. Instead they indicate that they are to be replaced with a valid expression or statement as appropriate, referring to other parts of this Manual for the correct construction of expressions and statements.

If an ellipsis (...) is used it indicates that the preceding syntactical item may be repeated (usually indefinitely). Exactly what part may be repeated is usually obvious from the context or the examples following. For example, the definition of a compound statement is:

BEGIN *statement; statement* ... END

In this case zero or more statements may appear between the BEGIN and the END. If more than one statement appears they are separated by semicolons.

PROGRAM

A G-Pascal program consists of a 'block' followed by a period. A block is explained below. For example, a simple program is:

```
BEGIN
  WRITE ("Hello")
END.
```

BLOCK

A block consists of the following:

CONST *constant-declarations*

VAR *variable-declarations*

PROCEDURE }
FUNCTION } *procedure and function declarations*

BEGIN *statements* END

The 'declarations' are all optional, however if used they must appear in the above order. Since a valid statement is a 'null' statement (i.e. nothing) then the minimum block is: BEGIN END

CONST

The constant declarations are used to assign a name to a constant, that is, a value that will not change during the execution of the program. The form of the constant declaration is:

CONST *identifier* = *constant*; ...

For example:

```
CONST cr = 13;
  clearscreen = 147;
  numberofplayers = 2;
  true = 1;
  false = 0;
  on = true;
  off = false;
```

As in the above example, a constant declaration can use an identifier which has already been declared as a constant (e.g. on = true;).

VAR

Variable declarations are used to name the variable data. Variables are either INTEGER or CHAR type, and may optionally be an array. Multiple variables of the same type may be separated by commas. This is the general format of variable declarations:

VAR *list-of-identifiers* : INTEGER;
list-of-identifiers : CHAR;
list-of-identifiers : ARRAY [*array-size*] OF INTEGER;
list-of-identifiers : ARRAY [*array-size*] OF CHAR;

The list-of-identifiers consists of one or more variable names, separated by commas.

The array-size is expressed as the maximum occurrence number of that array item. Arrays start at subscript zero, so that when you say ARRAY [10] you are referring to an array of 11 items, numbered 0 to 10.

Some examples:

```
VAR a, b, c, d, e, f : INTEGER;
  enemystrength,
  enemysize,
  enemydexterity : ARRAY [ 10 ] OF INTEGER;
  inputline : ARRAY [80] OF CHAR;
  ch : CHAR;
  amount1, amount2,
  amount3, amount4
  : INTEGER;
```

INTEGER variables are three bytes long each (in other words, three bytes of memory are reserved by the compiler for the contents of each variable or array element in the case of INTEGER arrays). This means that INTEGER variables can contain a number in the range -8388608 to +8388607. (In hexadecimal the range is: $0 to $FFFFFF). Integer variables can also hold up to a three-byte 'string' of characters, for example: "hi" or "ZZZ".

CHAR variables are one byte long each. This means that CHAR variables can contain a number in the range 0 to 255 (hexadecimal $0 to $FF). They can also contain a single character string, for example: "b" or "5".

The Compiler does not carry out 'type-checking' so that INTEGER and CHAR variables can generally be intermixed in expressions and elsewhere without problem. However, normally INTEGER variables are used to hold numbers, and CHAR variables are used to hold characters, particularly arrays of characters. If the result of an expression is stored in a CHAR variable and the result exceeds 255 (the maximum that it can hold) then the result is truncated – only the low-order byte is stored. Mathematically this means that the result stored is the number modulus 256.

G-Pascal does not support other data types, nor the TYPE declaration. However see the section 'Converting from other Pascals' for hints on how to simulate other data types in G-Pascal.

PROCEDURE and FUNCTION

Procedures and Functions are 'subroutines' which may be called later on in the program to achieve a specific purpose. The format of Procedure and Function declarations is identical. The difference between them is that functions return a value, and thus form part of expressions (explained later), whereas procedures do not return a value, and do not form part of expressions. The format of procedure and function declarations is:

PROCEDURE *procedure-name (argument-list); block ;*

FUNCTION *function-name (argument-list); block ;*

The '(argument-list)' is optional, however if used it consists of one or more arguments, separated by commas, inside parentheses. These are known as 'formal arguments'. In the body of the procedure or function the formal arguments are used as if they had occurred in a VAR declaration (of type INTEGER), although in fact no declaration is needed for formal arguments. During the execution of the program, when the procedure or function is invoked arguments are supplied (the actual arguments) – copies of these are then used by the procedure or function. For example, a function to double a number would have one argument (the number which is to be doubled) – the value returned by the function would be the double of the formal argument. (See example below).

You will note that the definition of a procedure or function declaration includes a 'block', whereas procedure and function declarations are themselves an (optional) part of a block. This is known as a recursive definition. In other words, a block may contain a procedure declaration which itself contains a block, which in turn may contain another procedure declaration and so on. This means that procedures and functions may themselves contain CONST, VAR, PROCEDURE and FUNCTION declarations.

If the procedure or function contains its own VAR declarations then these are known as 'local' variables as they can only be referred to inside the procedure or function in which they are declared. Outside the procedure or function they have no meaning and may not be referred to (this is referred to as the 'scope' of that declaration). It is good programming practice to use local variables for data that is not needed outside the procedure or function, such as loop counters, flags and so on. This avoids possible unintentional conflict with other variables used elsewhere.

The values of local variables are lost when the procedure or function in which they are declared is exited. That is, when a procedure or function is invoked the contents of all of its local variables are not defined – they are not retained from any previous invocation. They may only be used while the procedure or function is active, and are discarded when the end of the procedure or function is reached.

Examples of Procedures and Functions

Here are some examples of procedure and function declarations:

```
PROCEDURE clearscreen;
CONST home = 147;
BEGIN
WRITE ( CHR(home) )
END; (* of clearscreen *)

PROCEDURE printdoubles ( firstnumber, lastnumber );
VAR number : INTEGER;

FUNCTION double ( x );
BEGIN
double := x * 2
END; (* of double *)
```

```
BEGIN
FOR number := firstnumber TO lastnumber DO
   WRITELN ( number, " times 2 is ", double ( number ))
END; (* printdoubles *)

BEGIN (* main program *)

clearscreen;
printdoubles (20,30)

END. (* of main program *)
```

The above example, as well as illustrating procedures and functions, is a complete program which you may want to key in and try for yourself. Note that the function 'double' is declared within the procedure 'printdoubles'. This is an example of a 'nested' function. Procedures and functions may be nested indefinitely. Notice the different way that procedure and functions are invoked (called). Procedures are invoked by naming them as a statement. The main program above in fact merely consists of two procedure invocations. Functions however are invoked by naming them as part of an 'expression' – in this case the function 'double' forms a part of the WRITELN statement.

The function 'double' also illustrates how a function can use its parameters (arguments) in a calculation. When 'double' is called it is passed the parameter 'number' (i.e. double (number)). A copy of the value contained in 'number' is made and passed to 'double' which then considers it to be its argument 'x'. 'x' is then multiplied by two and assigned to the name of the function, which is the way that the function returns its result to the part of the program that invoked it. (i.e. double := x * 2).

Recursive Procedures and Functions

Procedures and functions can also invoke themselves recursively which can sometimes be useful. For example, here is a way of calculating factorials by recursion:

```
VAR number : INTEGER;

FUNCTION factorial ( x );
BEGIN
  IF x = 1 THEN
    factorial := x
  ELSE
    factorial := x * factorial (x – 1)
END; (* factorial *)

BEGIN (* main program *)
  FOR number := 1 TO 10 DO
  WRITELN (number, "! = ", factorial (number))
END.
```

The example above is also a complete program which you may wish to try for yourself. The function 'factorial' keeps calling itself (recursing) until it can go no further (1 factorial is 1). There are other ways of calculating factorials but this example illustrates recursion quite well.

(A factorial of an integer is the product of all the integers from 1 to itself. e.g. 4 factorial, which is represented mathematically as 4!, is 4 * 3 * 2 * 1, which is 24. 5 factorial is 120 and so on.)

STATEMENTS

The parts of the program (or block) that 'do something' are the statements. The beginning of the statements part of a block is signalled by the word BEGIN and the end by the word END. Like the definition of a block, the definition of a statement is recursive – that is, some statements may contain other statements. The simplest example of this is the 'compound statement'. A compound statement looks like this:

BEGIN
statement; statement; statement ...
END

In other words, wherever a statement is allowed, the word BEGIN may be placed, followed by any number of statements separated by semi- colons, followed by the word END. For example, the definition of an IF statement is as follows:

IF *expression* THEN *statement* *ELSE statement*

The following examples are valid IF statements:

```
IF x = 3 THEN
  b := 5
ELSE
  b := 10;

IF word = "abc" THEN
  BEGIN
  a := 1;
  b := 2
  END
ELSE
  BEGIN
  a := 5;
  b := 10
  END;
```

Null statements

A valid statement is the 'null' statement (in other words, nothing). So the following is also a valid (although rather meaningless) IF statement:

```
IF a = 21 THEN ELSE;
```

The absence of any symbols between the THEN and the ELSE and also after the ELSE indicates the null statement. In other words, nothing will happen. Occasionally this is useful – for example, during debugging you may temporarily convert a statement to comments (by enclosing it in (* and *)) – this will have no ill effect.

We now summarise the various G-Pascal statement formats, with a brief description of their purpose.

Assignment statement

variable := *expression*

The assignment statement evaluates the expression (expressions are explained later) and places the result in the named variable. e.g.

```
A := B * 25;
```

Procedure invocation

procedure-name (expression, expression ...)

By naming a previously defined procedure you invoke it – that is, transfer control to the procedure to carry out the statements defined within that procedure's declaration. If the procedure declaration contained a list of formal arguments then the actual values to be passed to that procedure are now supplied in the form of a list of expressions, separated by commas, all in parentheses. e.g. PLAYANOTE (PITCH, DURATION); If no arguments are required then the parentheses are not required. e.g. NEXTGAME;

IF

IF *expression* THEN *statement* ELSE *statement*

The expression is evaluated. If it is true (not zero) then the first statement is executed, if it is false (zero) then the second statement is executed. The 'ELSE statement' clause is optional. (Example previously).

WHILE

WHILE *expression* DO *statement*

The expression is evaluated. If it is true (non-zero) the statement is executed. This process is repeated until the expression becomes false (zero).

e.g.

```
x := 0;
   WHILE x < 10 DO
     BEGIN
     x := x + 1;
     WRITELN (x," squared is ",x * x)
     END
```

REPEAT

REPEAT *statement ; statement ...* UNTIL *expression*

The statements between REPEAT and UNTIL are executed. Then the expression is evaluated. If it is false (zero) this process is repeated. e.g.

```
REPEAT x := x + 1; y = y + 5 UNTIL x = 50
```

CASE

CASE *expression* OF
label-list1 : *statement1; ...*
label-list999 : *statement999*
ELSE *statement*
END

The expression is evaluated. This is called the 'case selector'. It is then compared in turn to the lists of expressions (labels) preceding each statement. If it matches one of them then the statement following the colon is executed and the CASE statement terminates. If no label matches the selector then the statement following the ELSE is executed. As in the IF statement the ELSE clause is optional. The label list consists of one or more expressions, separated by commas. This is an extension of standard Pascal which only allows a list of constants. e.g.

```
CASE actionnumber OF
  1    : processaction1;
  2, 3, 4 : processactions234;
  5, 6 : processactions56
ELSE   processerroraction
END
```

The END is part of the CASE statement and should not be confused with the normal BEGIN END pairs normally found elsewhere in Pascal.

Here is an example of using expressions in a CASE statement:

```
          CASE true OF
  (x > 0) AND (x < 10) : smallx;
  (x >= 10) AND (x < 100) : mediumx;
  x >= 100          : largex
ELSE            toosmallx
END
```

WRITE and WRITELN

WRITE (*output-list*)
WRITELN (*output-list*)
WRITELN

WRITE and WRITELN function identically except that WRITELN automatically writes a 'carriage return' at the end of the output- list. In other words, at the end of a WRITELN the cursor (or printer if printing) commences a new line. WRITELN without any arguments just writes a carriage return.

The output list is evaluated and written to the screen. The output list consists of one or more of: expressions, string constants, or the special functions HEX and CHR, separated by commas.

String constants are written 'as is', e.g. WRITE ("hello") would display the word 'hello' on the screen. If you want the quote symbol to appear in the string itself then two consecutive quotes should be used. For example, to display: MY NAME IS "SUE" you would enter: WRITELN ("MY NAME IS ""SUE""").

If an expression is written on its own it is evaluated and the result displayed as a decimal number. e.g. WRITE (5 * 6) would display '30' on the screen.

To write 'screen control' codes (such as clear screen, shift to upper-case and graphics and so on) the number should be written as: CHR (x) where x is an expression. e.g. WRITELN (CHR(147)) would clear the screen. (This is the same as: PRINT CHR$(147) in Basic).

To write a number in hexadecimal the number should be written as: HEX (x) where x is an expression. e.g. WRITE (HEX(10)) would display '00000A'.

The functions HEX and CHR are only meaningful within a WRITE or WRITELN statement.

```
WRITELN (65," in hex is ",hex(65)," and in ascii is ",chr(65));
```

READ (*input-list*);

The input list consists of a list of one or more variables, separated by commas. Each item in the list is treated separately, as if they had appeared in separate READ statements. There is no built-in method of accepting a list of numbers from one line of input, although you could write a procedure to do this by accepting a string and decoding it as desired. The effects of the various types of input-list variables are as follows:

a) Variable of type CHAR – a single character is accepted from the keyboard and placed in the named variable. The system does not wait for the RETURN key to be pressed. The character is not echoed on the screen.

b) Array of type CHAR with no subscript supplied – a line of input is accepted from the keyboard and placed in the named array, starting at subscript zero. The line is echoed on the screen as it is typed in, and the normal cursor control keys may be used to edit the line prior to the RETURN key being pressed. If the line entered is smaller than the array size then a RETURN symbol will appear in the array (i.e. the value 13) after the last character entered. *Warning – the Commodore 64's full-screen editor will cause text on the screen to the right of the cursor to be considered part of the input, even if it wasn't actually typed in at that time.* This means that a 'default' answer can be displayed and the cursor positioned to the start of it. If the user just presses RETURN then the text to the right of the cursor will be returned automatically.

c) Variable of type INTEGER – a line of input is accepted from the keyboard. The line is echoed on the screen as it is typed in, and the normal cursor control keys may be used to edit the line prior to the RETURN key being pressed. After RETURN is pressed an attempt is made to interpret it as a decimal number and place the result in the named variable. The first character must be a number or minus sign, the following characters must be numbers, until the RETURN is pressed. If these conditions are not met the built-in function INVALID is set true, otherwise (if the number is OK) INVALID is false. *Warning - the Commodore 64's full-screen editor will cause text on the screen to the right of the cursor to be considered part of the input, even if it wasn't actually typed in at that time.* This means that a 'default' answer can be displayed and the cursor positioned to the start of it. If the user just presses RETURN then the text to the right of the cursor will be returned automatically.

d) Variable of type INTEGER followed by a $ symbol. A line of input is accepted from the keyboard. The line is echoed on the screen as it is typed in, and the normal cursor control keys may be used to edit the line prior to the RETURN key being pressed. After RETURN is pressed an attempt is made to interpret it as a hexadecimal number and place the result in the named variable. The numeric string input must be from 1 to 6 hexadecimal digits for the attempt to succeed. If the number is not recognized as a hexadecimal number then the built-in function INVALID is set true, otherwise (if the number is OK) then INVALID is false.

Some examples:

```
VAR number, hexnumber : INTEGER;
  character : CHAR;
  line    : ARRAY [80] OF CHAR;
BEGIN
READ (number);    (* read a decimal number *)
IF INVALID THEN WRITELN ("bad number");
READ (hexnumber $ ); (* read a hex number *)
IF INVALID THEN WRITELN ("bad hex number");
READ (character);    (* read a single character *)
READ (line);        (* read a line into an array *)
END.
```

FOR

FOR *variable* := *expression1* TO *expression2* DO *statement*

Expression1 is evaluated and assigned to the variable. It is then compared to expression2. If it is less than or equal to expression2 then the statement is executed. Then 1 is added to the variable, and it is compared to expression2 again and so on, until the variable is greater than expression2.

FOR *variable* := *expression1* DOWNTO *expression2* DO *statement*

Expression1 is evaluated and assigned to the variable. It is then compared to expression2. If it is greater than or equal to expression2 then the statement is executed. Then one is subtracted from the variable, and it is compared to expression2 again and so on, until the variable is less than expression2.

For example:

```
FOR j := 1 TO 20 DO
  WRITELN (j);

FOR k := 100 DOWNTO 20 DO
  WRITELN (k)
```

CALL

CALL (*expression*)

The CALL statement transfers control to the machine-code subroutine located at the address which the expression evaluates to. For example, to call the 'close all files' routine in the Monitor Kernal (at address $FFE7) you would say:

```
CALL ( $FFE7 );
```

It is possible to set up the A, X, Y registers and condition codes before the CALL and examine them after the CALL See the section on 'Machine Language Subroutines' for more details.

Other statements

There are other statements with specialized uses which are G- Pascal extensions for the Commodore 64, which control the cursor, graphics, sound effects, clock etc. These are described in more detail in other sections of this Manual.

EXPRESSIONS

An expression (or 'arithmetic expression') forms part of the definition of many statements, as seen previously. For example, an assignment statement assigns the result of the evaluation of an expression to a variable. Like the definitions for a block and a statement, the definition of an expression is recursive, that is, an expression can contain another expression. Expressions consist of a combination of operands and operators. e.g.

```
5
A < 2
(a * b / c) + 14
```

Operands

array-name [index-expression]
constant
function-name
function-name (argument-list)
variable
(expression)

Operators

(In decreasing order of precedence):

NOT
* / DIV MOD AND SHL SHR
+ – OR XOR
= <> < > <= >=

Precedence controls in which order the expression is evaluated. Since * is on a higher precedence than + then the expression:

```
5 + 8 * 10
```

evaluates to 85, not 130, since the multiplication is done before the addition. Where operations of equal precedence are presented then they are evaluated from left to right. Since the operators AND and OR are on a higher precedence than the relational operators (=, <, › and so on) then conditional expressions generally require parentheses around expressions combined by AND and OR. For example:

```
IF ( a > 5 ) OR ( b < 10 ) THEN<
```

If the parentheses were omitted G-Pascal would evaluate '5 OR b' before doing the relational tests, which would probably not have the desired result.

MEANING OF OPERATORS

NOT

NOT reverses the value of a boolean operand. Unlike all other operators which are inserted *between* two operands (e.g. in the expression: 4 + 6 the operator '+' comes between the '4' and the '6'), NOT *precedes* a single operand. If its operand is true (non-zero) then NOT makes it false (zero). If its operand is false (zero) then NOT makes it true (1). e.g.

```
REPEAT game UNTIL NOT alive;
```

*

* is the symbol for multiplication. It multiplies two numbers together. e.g.

```
salary := wage * 52;
```

/ and DIV

/ and DIV are synonymous and are the symbols for division. They result in integer division. (If the division cannot be performed exactly then the remainder is discarded). e.g.

```
inches := feet / 12;
```

The remainder of a division can however be established with the MOD operator.

MOD

MOD returns the remainder of a division (often known as the *modulus* – hence its name). e.g.

```
diceroll := RANDOM MOD 6 + 1;
```

AND

AND performs a boolean 'and' of each of the 24 bits in the two operands. That is, a result bit is on only if both corresponding bits in the operands are on. The normal use for AND is in the conventional sense of checking that two conditions are true simultaneously, however it can also be used to 'mask' out unwanted bits in an integer. e.g.

```
IF (a > 4) AND (b < 5) THEN     (* check both conditions are true *)
IF PADDLE (2) AND $FF THEN      (* isolate low order byte *)
```

SHL

SHL performs a 'shift left' operation. It takes the first operand and shifts it left the number of bits specified in the second operand (maximum of 24 shifts). Each shift left is equivalent to multiplying the first operand by 2, so SHL is a simple and quick way of raising a number to a power of 2. For example, to multiply a number by 2 to the power 6:

```
result := number SHL 6;
```

SHR

SHR performs a 'shift right' operation. It takes the first operand and shifts it right the number of bits specified in the second operand (maximum of 24 shifts). Each shift right is equivalent to dividing the first operand by 2, so SHR is a simple and quick way of dividing a number by a power of 2. For example, to divide a number by 2 to the power 6:

```
result := number SHR 6;
```

+

+ is the symbol for addition. It adds its two operands together. e.g.

```
result := a + b;
```

–

– is the symbol for subtraction. It subtracts the second operand from the first. e.g.

```
klingons := klingons – 1;
```

OR

OR performs a boolean inclusive 'or' of each of the 24 bits in the two operands. That is, a result bit is on if either or both of the corresponding bits in the operands are on. The normal use for OR is in the conventional sense of checking that either of two conditions are true, however it can also be used to 'turn on' certain bits in an integer. e.g.

```
IF (a > 4) OR (b < 5) THEN    (* check either condition is true *)
mask := mask OR 1;        (* turn on low order bit *)
```

XOR

XOR performs a boolean exclusive 'or' of each of the 24 bits in the two operands. That is, a result bit is on if either of the corresponding bits in the operands are on *but not both*. XOR is a G-Pascal extension and is less likely to be used than OR, however one important use for XOR is for toggling a bit – that is, turning it off if it is on, or turning it on if it is off. This can be useful when defining your own character sets – the 'inverse' of any character can be obtained by XORing the character with $FF. This will switch any 1's to 0's, and any 0's to 1's. e.g.

```
b := b XOR 1; (* toggle contents of b *)
byte := byte XOR $FF; (* invert contents of byte *)
```

= <> < > <= >=

The above symbols are all 'relational operators'. That is, they return a boolean value (true or false) depending on whether or not the relation is true. The result of a relational operation will always be zero (false) or 1 (true). Relational operators are usually used in IF, WHILE or REPEAT statements, however they can be used anywhere that an expression is permitted. e.g.

```
IF ships < 3 THEN .
IF answer = 6 THEN
result := b >= 4;
```

NOTES ON THE COMPILER

ARRAYS

Single dimension arrays only are permitted. If multi-dimensional arrays are needed it is easy to write functions to simulate them. For example, if you want an 8 by 15 array, declare your array as having 120 elements (8 times 15) and write a function that multiplies the first subscript by 8 and adds the second subscript. That way each combination of subscripts 'maps' onto a unique element.

Subscripts must be enclosed within square brackets ([and]). These are located above the colon and semicolon (: and ;) on the keyboard.

Range checking of arrays

No range checking is carried out on any arrays. This means that if you declare an array of 10 items and write to the 11th upwards, or to the –1st downwards then you will 'clobber' something unexpected. This may cause your program to 'crash' or behave strangely, or quite possibly the entire G-Pascal system may crash.

Part of the reason for this is that 'return' addresses for procedure or function calls, and also 'stack frame addresses' are stored on the stack, right next to your variables. Therefore if an array is defined as the first variable within a procedure or function then exceeding the bounds of a declared array by just *one* element could make the procedure or function lose track of where to return to. *If you think an array variable may exceed its declared bounds build a check into your program to make sure it doesn't.*

Array allocation

Arrays are allocated *downwards* in memory. This is important if you are storing a machine language program in an array, for example. In a given array, the highest array element has the lowest memory address.

'PROGRAM' STATEMENTS

Programs do not start with the word PROGRAM. A program must start with one of: CONST, VAR, PROCEDURE, FUNCTION or BEGIN. Failing this, the error message 'BEGIN expected' will appear.

CONSTANTS

Constants (a constant is a factor in an expression that does not change throughout the execution of a program) can be expressed in one of four ways interchangeably:

1. Decimal constants: for example: 1096, –33, +99

Decimal constants are simply signed numbers to the base 10. They may range from –8388608 to +8388607.

Warning – since the compiler interprets a plus or minus sign directly in front of a number as being the 'sign' of that number (e.g. –5) then when using plus or minus signs for arithmetic (addition and subtraction) then at least one space must follow the sign if the next symbol is a number. In other words:

```
a := b+1; would
```

give an error because the '+' would be interpreted as the sign of '1', not an addition operation. The correct way to represent the above would be:

```
a := b + 1;
```

2. Hexadecimal constants, for example: $A1, $FFC.

Hexadecimal constants are unsigned numbers to the base 16. They may range from $0 to $FFFFFF. The '$' is required.

3. String constants, for example: "Z", "NO", "YES".

String constants are a string of between one and three characters, enclosed in quotation marks. If only one or two characters appear in the string then the high-order byte/s are set to zero. Therefore when comparing to, or moving to or from, a CHAR type variable only one character should appear between the quotes as CHAR variables are only one byte long. The use of two or three character string constants would primarily be restricted to applications which involve a lot of 'word' work, such as Adventure-type games or other applications which involve three letter commands.

A string constant "abc" will be stored as:

```
"a" + "b" * 256 + "c" * 65536
```

or in other words:

```
"a" OR "b" SHL 8 OR "c" SHL 16
```

4. Identifiers which appear in a CONST declaration.

For example, if the declaration: CONST cr = 13;
appears in the program then 'cr' is considered to be a constant. The use of CONST declarations for constants is *highly recommended* as they make the program much more readable and easy to follow. For example: WRITE (chr(147)) and WRITE (chr(clearscreen)) will both clear the screen (if 'clearscreen' is declared to be equivalent to 147 in a CONST declaration) but the latter makes the intention much clearer when reading the program.

DATA TYPES

Integer

Integers are stored as 3-byte signed numbers and therefore range from –8388608 to 8388607.

Char

CHAR type variables are stored as one byte each. No type checking is carried out by the compiler so in fact INTEGER and CHAR variables may be used completely interchangeably in G-Pascal, except:

a) CHAR variables have a different meaning to integer variables in the READ procedure.

b) The result of an expression stored in a CHAR variable will be MOD 256 (that is only the low order byte will be stored – the high order two bytes will be discarded). Therefore a CHAR variable will always be in the range of 0 to 255 (or $0 to $FF).

c) CHAR variables only use one third the amount of memory to store as integers – particularly important in big arrays.

Other data types

Other data types are not supported, nor is the TYPE statement. (But see 'Converting from other Pascals' for hints on how to simulate other data types in G-Pascal).

ELSE CLAUSE

The ELSE clause may be used with the IF statement, in which case it is always associated with the most recent un-ELSE-ed IF.

The ELSE clause may also be used with the CASE statement to cause a statement to be executed if none of the labels agree with the case selector.

MEM ARRAY

(read this if you like peeking/pokeing)

The MEM array is a pre-defined integer array starting at address zero.
For example:

```
FRED := mem [ $5000 ];
```

would place in FRED the 3 bytes in memory, starting at address $5000;

or:

```
mem [ $5000 ] := $ABCDEF;
```

would store $ABCDEF in memory addresses $5000 to $5002.

Caution: Careless use of MEM as a receiving field (as in the second example) could 'crash' your G-Pascal system or destroy your program. Use with care!

Another caution: Use of absolute addresses via MEM and MEMC makes your programs machine-dependent and not portable.

A third caution: The subscript of the MEM array is always interpreted as an absolute address so that MEM [0] and MEM [1] in fact overlap by two bytes – they both share addresses 1 and 2. This is unlike ordinary integer arrays where each 'occurrence' represents an address 3 bytes away from its neighbour.

MEMC ARRAY

The MEMC array is a pre-defined CHAR array starting at address zero. Use MEMC for peeking/pokeing single bytes.

ADDRESS

The run-time address of a variable can be established by the

```
ADDRESS (identifier)
```

function. For example, to find the address in memory of FRED:

```
J := ADDRESS ( FRED );
```

The ADDRESS function would only be of interest to more experienced programmers for special purposes such as calling machine code routines contained within an array, or passing the address of a variable to a procedure. It is unique to G-Pascal and therefore not portable.

LOAD

The LOAD statement will load a file from disk or cassette under program control. The LOAD statement takes the following form:

```
LOAD (device, address, flag, filename );
```

The 'device' represents the device number which is 1 for cassette and 8 for disk. The 'address' is the address to which the data is to be loaded. The 'flag' is 0 for a Load and 1 for a Verify. The 'filename' is a string containing the filename.

After the load, the built-in function INVALID should be checked. If it is zero then the load was OK. If it is non-zero then INVALID contains the error code.
e.g.

```
LOAD (8, $1000, 0, "FILEA");
IF INVALID THEN WRITELN ("ERROR", INVALID , "ON LOAD");
```

SAVE

The SAVE statement will save a file to disk or cassette under program control. The SAVE statement takes the following form:

```
SAVE (device, start-address, end-address, filename );
```

The 'device' represents the device number which is 1 for cassette and 8 for disk. The 'start-address' is the start address from which the data is to be saved. The 'end-address' is the end address of the block of data to be saved. The 'filename' is a string containing the filename.

If the start address is greater than or equal to the end address a run-time error will occur.

After the save, the built-in function INVALID should be checked. If it is zero then the save was OK. If it is non-zero then INVALID contains the error code. e.g.

```
SAVE (8, $1000, $1800, "FILEA");
IF INVALID THEN WRITELN ("ERROR ",INVALID, " ON SAVE");
```

SAVE may be used to save a block of variables from a program so that they can be loaded back again later on (for example, saving the current position in an adventure game). In this case remember that variables are allocated *downwards* on the stack, so that the lowest memory address is in fact the last variable declared. The safest way of doing this is to declare two 'dummy' variables to pinpoint each end of the variable data, like this:

```
var firstvar : char; (* first program variable – highest address *)
a,b,c,d,e,f : integer; (* all program variables go here *)
i,j,k,l,m : char;
(* and so on *)
lastvar : char; (* last program variable – lowest address *)
begin
save (8, address(lastvar), address(firstvar), "varfile");
load (8, address(lastvar), 0, "varfile");
end.
```

Of course, this technique will only work if the number of variables is the same between the SAVE and the LOAD, in other words, if the data declarations in the program are the same.

GETKEY

GETKEY is a function that indicates whether or not a key is being pressed on the keyboard (or is in the keyboard queue). If no key is currently being pressed it returns zero – if a key is being pressed it returns a value corresponding to that key.

To make a program wait until any key is pressed for example just say:

```
REPEAT UNTIL GETKEY;
```

However if you need to know what the value of the key is then it must be saved in a temporary variable. e.g.

```
REPEAT
  KEYVALUE := GETKEY;
UNTIL KEYVALUE;
CASE KEYVALUE OF ..... (* and so on *)
```

Normally you would use the READ statement if you want to wait until a key is pressed. GETKEY is intended for applications where the program wants to occasionally check for keys being pressed on the keyboard but do other things in the meanwhile if they are not.

ABS (value);

The built-in function ABS returns the absolute value of its argument. In other words, ABS will always return a positive result, whether or not the argument is positive. This can be handy for establishing the distance between two points, regardless of which one is greater than the other. e.g.

```
distance := ABS ( x1 – x2 );
```

SAMPLE PROGRAM

```
(* ERATOSTHENES SIEVE PRIME NUMBER GENERATOR   %L *)

CONST SIZE = 1000;
   TRUE = 1;
   FALSE = 0;
   HOME = 147;
   PERLINE = 5;

VAR FLAGS : ARRAY [SIZE] OF CHAR;
I,PRIME,K,COUNT,ONLINE : INTEGER;

BEGIN
COUNT := 0;
WRITELN (CHR(HOME));
ONLINE := 0;
FOR I := 0 TO SIZE DO
   FLAGS [I] := TRUE;
FOR I := 0 TO SIZE DO
   IF FLAGS [I] THEN
   BEGIN
      PRIME := I + I + 3;
      K := I + PRIME;
      WHILE K <= SIZE DO
      BEGIN
      FLAGS [K] := FALSE;
      K := K + PRIME
      END;
      IF ONLINE > PERLINE THEN
      BEGIN
      WRITELN; (* NEW LINE *)
      ONLINE := 0
      END;
      ONLINE := ONLINE + 1;

      COUNT := COUNT + 1;
      WRITE (PRIME," ")
   END;
   WRITELN;    WRITELN (COUNT, " PRIMES")
END.
```

COMPILING YOUR PROGRAM

Compile

If you select 'Compile' from the Main Menu or Editor your program will be compiled and converted to P-codes, ready for Running. You may select a listing of the program to appear during compilation, or change the address at which the P-codes are located by using the Compiler Directives described below. If the compile is successful (no errors) you may immediately press 'R' (for Run) to test your program.

Syntax

The 'Syntax' option also compiles your program, however it does not generate P-codes so you must do a Compile afterwards if you have no errors and wish to run your program. As P-codes are not generated the %P compiler directive (display P-codes) does not function (acts the same as %L). In all other respects Syntax and Compile are identical. The main use for 'Syntax' is to check for errors if the P- codes are going to clobber the source code during the compilation process (this can only happen if the %A compiler directive is used). In this case, use the 'Syntax' option to check that the program has no errors, then save it to disk or cassette, then Compile it.

Asterisks

If you do not request a listing an '*' is displayed on the screen as every 32 source program lines are compiled. This is to reassure you that 'something is happening', and give a visual indication of how fast (and how far) the compilation is proceeding.

Errors

If there is an error in your program the compilation will stop with an arrow pointing to the symbol being processed when the error was detected (this not necessarily being the actual cause of the error as such), the word '*** Error' and an English error message. See the section 'Compiler Error Messages' for details about the meaning of errors.

COMPILER DIRECTIVES

Compiler directives are special symbols inserted within comments to cause G-Pascal to take special action during the compilation, such as producing a listing or placing the P-codes at a different address.

Compiler directives must appear within comments (i.e. after a (* symbol, and before its corresponding *) symbol) as they are not part of the Pascal language as such. They consist of the percentage symbol (%) directly followed by a code letter indicating the type of directive. The code letter may be in upper or lower case. These are described below.

%L (Listing)

The %L directive causes a listing of the program (with the P-code addresses displayed on the left in parentheses) commencing with the line on which the %L directive appears. The listing is the same as an Editor List, except for the P-codes on the left. At times knowing the P-code addresses is useful, particularly when a run-time error occurs (such as Divide by zero). In this case the address displayed when the run-time error occurs can be used to locate the line in the program which caused the error.

The %L directive would normally appear in the first line of the program, however it may be placed further down if only a partial listing is desired. Listing may be turned off with the %N directive.

```
(* %L *)
```

%N (No listing)

The %N directive stops the compiler from listing the program as it compiles. Instead, an asterisk is displayed as each 32 program lines are compiled. This is the default.

```
(* %N *)
```

%P (P-code listing)

The %P directive causes the compiler to list the program, and also to list each P-code that is generated for each statement. This directive would not normally be used, unless you are interested in what P-codes are generated for each statement. See 'Meanings of P- codes' for a description of what each P-code means. Like the %L directive %P is cancelled with an %N directive.

```
(* %P *)
```

%A (Address of P-codes)

The %A directive defines where the P-codes are to be placed during the compilation process. It should be followed by a decimal or hexadecimal address or the error 'Constant expected' will appear.

It is essential that the %A directive appear at the very start of the program, before the first CONST, VAR, PROCEDURE, FUNCTION or BEGIN. If this is not done the P-codes will not be compiled contiguously and a run-time error (or worse) will happen when the program is run.

If the %A directive is omitted then the P-codes will be placed directly following the end of the source program. This is the usual and recommended method of compiling. The %A directive should only be used if the 'Memory Full' error appears during compilation (which means that there is insufficient room at the end of the source program for the P-codes), or when compiling independent modules.

The address for the P-codes must be chosen with care. The permitted range is $800 (just after screen memory) to $4000 (overlapping the source program). Outside this range the error message 'Number out of range' will appear. A P-code address of $800 would be sensible for many applications, however care must be taken that DEFINESPRITE statements and bit-mapped graphics do not clash with the P-codes. See the 'Memory Map' section in this Manual for more details about what memory addresses may be used for which purpose.

In the event that the P-codes are placed at $4000 (the start of the source program), or just below, then the compilation process will cause the source program to be replaced by the P-codes, thus *effectively destroying the source program. Under these circumstances it is essential that the source program be saved to disk or cassette before a Compile, otherwise the source program will be lost.*

As the compiler is a single-pass compiler the technique of 'clobbering' the source program with the P-codes is an effective method of making maximum use of available memory, however there is the inconvenience of having to re-load the program from disk or cassette after each Compile. Remember that the 'Syntax' option is a method of compiling the program without generating P-codes, and should be used to ensure that there are no compile errors before the final compilation which does generate P-codes is done.

Examples of the %A option:

```
(* %A $800 – place P-codes after screen memory *)

(* %A $4000 – overwrite source code with P-codes *)
```

COMPILER ERROR MESSAGES

If the compiler detects an error in the G-Pascal program it will return one of 36 error messages. The messages appear in English, as given below. An upwards arrow will point to the symbol currently being processed when the error was detected. If the compiler has detected the end of the program unexpectedly the arrow will point to the last symbol in the program (for example if the final period (.) is missing).

In some cases the error will be in the symbol that the arrow is actually pointing to (for example, 'Undeclared Identifier' or 'Number out of range'). However in a lot of cases the error will be an error of omission (For example, the message '; expected' usually means that a semicolon has been omitted from the end of the previous line). In these cases the error message usually refers to some problem in the previous statement or clause. Therefore, if the meaning of an error is not obvious, *list and examine carefully the last ten or so lines prior to and including the line containing the error.*

All errors are fatal – in other words as soon as an error occurs the compilation is halted and the compiler automatically returns to the Editor so that the error can be corrected. As G-Pascal compiles very rapidly detection and correction of errors is a quick and easy process.

The list of error messages below is intended as a guide to the usual circumstances surrounding a given error – they should help in understanding a particular error.

As the error messages are in upper and lower case the compiler automatically switches the character set to upper and lower case before displaying errors.

= expected

The compiler is processing a CONST declaration and is expecting an '=' sign to come between the constant name and its value. (e.g. CONST TRUE = 1;).

:= expected

The compiler is processing an assignment statement or a FOR statement and is expecting a ':=' to follow the variable name. (e.g. K : 1;).

; expected

The compiler is expecting a semicolon. It is probably missing from the end of the previous statement.

; or END expected

The compiler is processing a compound statement – that is, one beginning with a BEGIN and ending with an END. It has come to the end of a statement and now expects either a semicolon followed by another statement, or the word END.

, expected

The compiler is processing a list of arguments and now expects a comma, followed by another argument. (e.g. CURSOR (6, 7);).

, or : expected

The compiler is processing a CASE statement and is expecting a comma or colon to follow the case selector, or the compiler is processing a VAR declaration and is expecting a comma or colon to follow the previous identifier. (e.g. VAR X, Y, Z : INTEGER;);

(expected

The compiler is processing a statement that requires arguments to be supplied in round brackets (for example, arguments to a user- declared procedure or function, or arguments to a built-in procedure such as CURSOR). However it cannot find the opening parenthesis.

) expected

The compiler has finished processing a list of arguments and now expects a closing parenthesis, however it cannot find one. This error may appear if an argument is mistyped or a comma which should separate arguments is omitted.

[expected

The compiler is processing an array name (or an array declaration) and now expects a subscript inside square brackets.

] expected

The compiler has finished processing an array subscript and now expects the closing square bracket.

***) expected**

The compiler has reached the end of the program but is still in the middle of processing a comment. The probable cause of this is that in the program a comment has commenced with '(*' but was not terminated with its corresponding '*)'.

. expected

There is no period following the final END in the program.

BEGIN expected

The compiler is processing a block and now expects the word BEGIN to mark the start of the statements in that block. If they have not already been declared this message also means that the compiler will also accept CONST, VAR, PROCEDURE or FUNCTION declarations. This error is usually caused by either forgetting to put the word BEGIN before the statements in a Procedure, Function or main program, or by making a mistake in the CONST or VAR declarations (such as misspelling CONST for example).

Compiler limits exceeded

It is unlikely this error will appear. However if it does simplifiy the statement that it is currently processing.

Constant expected

The compiler is expecting a constant, in other words one of: a number (such as 20), a hex constant (such as $ABCD), a string constant (such as "xyz"), or an identifier declared as a constant in a CONST declaration.

Data type not recognised

The compiler is processing a VAR declaration and has not found either the word INTEGER or CHAR.

DO expected

The compiler is processing a WHILE statement and now expects the word DO. (e.g. WHILE X > 50 DO ...).

Duplicate identifier

The program is attempting to use the same name twice for an identifier under illegal circumstances. It is permitted to use the same name twice if one occurrence is a 'global' declaration (at the start of the program) and another declaration is 'local' (inside a procedure or function). The same name cannot be used twice within the same 'group' of declarations, for example VAR FRED, FRED : INTEGER; would be an error.

Identifier expected

The compiler is processing a CONST, VAR, PROCEDURE, or FUNCTION declaration, or the argument to the ADDRESS function, and is now expecting a user-supplied identifier. (e.g. VAR FRED).

Illegal factor

The compiler is processing an expression and finds an illegal factor. This could be caused by an illegally constructed arithmetic expression (e.g. 5 + *), a missing argument to a procedure or function (e.g. WRITE ()), or a missing expression where one is expected (e.g. IF THEN).

Illegal identifier

An identifier has occurred in a context in which it was not expected.

Incorrect string

The compiler has detected that a string literal is not where it should be or is not terminated. In the case of the LOAD and SAVE statements this error will occur if there is no file name where one is expected, otherwise the error is caused by an opening quote symbol on a line but no corresponding closing quote symbol. (e.g. "HELLO). In this example the upwards arrow would point to the 'H' in 'HELLO'.

Incorrect symbol

The compiler believes it is at the end of the program, however it has found more program following the final period.

Literal string of zero length

A string of zero length (i.e. two consecutive quote symbols: "") was encountered.

Memory full

There is insufficient room for your P-codes to follow the source program. Reduce the size of your program, or place the P-codes at a different address than the default one of at the end of the source program. See the section on the %A compiler directive for details on how to do this.

Number out of range

The compiler is processing a decimal or hexadecimal literal and has found that it is too large (or too small). The allowable range for decimal integers is –8388608 to +8388607. The allowable range for hexadecimal integers is $0 to $FFFFFF.

OF expected

The compiler is processing a CASE statement and now expects the word OF. (e.g. CASE recordtype OF ...).

Parameters mismatched

The compiler is processing the invocation of a user-declared Procedure or Function and has detected that the number of arguments supplied to the Procedure or Function invocation does not agree with the number of arguments declared for the Procedure or Function.

Stack full

The compiler's internal stack has overflowed due to processing too many nested procedures, functions or expressions. This message is very rare, however if it does occur the problem can be corrected by re-writing the program with less 'nesting'. (A nested procedure, for example, is a procedure within a procedure within a procedure etc.).

String literal too big

The compiler has encountered a string literal in an expression, however it is more than 3 characters long. String literals of more than 3 characters are only allowed in the LOAD, SAVE and WRITE statements.

Symbol table full

The program has too many user-declared identifiers (in other words, CONST, VAR, PROCEDURE or FUNCTION names). There are three possible solutions to this problem. First, reduce the number of variables if possible. Second, reduce the length of identifiers (with the Editor's Replace command). Third, make identifiers 'local' to procedures or functions as much as possible. Local identifiers (in other words, CONST, VAR, PROCEDURE or FUNCTION declarations which are *inside* other procedures or functions) only occupy room in the symbol table whilst that procedure or function is being processed.

Each user-declared identifier occupies 10 bytes on the symbol table plus the length of the identifier. In other words, the word FRED would occupy 14 bytes on the symbol table – 10 plus 4 letters in 'FRED'. The total length of the symbol table is 4096 bytes so there is room for 256 identifiers if each is 6 characters long.

THEN expected

The compiler is processing an IF statement and now expects a 'THEN' to follow the conditional expression. (e.g. IF A = 5 THEN ...).

TO or DOWNTO expected

The compiler is processing a FOR statement and now expects the words TO or DOWNTO. (e.g. FOR x := 1 TO 10 DO ...).

Type mismatch

This error rarely occurs because G-Pascal allows mixed type operations in general. For example you may assign a integer variable to a char variable. However this error will occur if the program either: a) attempts to read a string into an integer array, or b) attempts to use a constant name on the left-hand side of an assignment statement.

Undeclared identifier

The identifier to which the upwards arrow is pointing has not been declared. Possibly it is misspelt, or no CONST, VAR, PROCEDURE or FUNCTION declaration exists for it. *All identifiers MUST be declared before they are referenced, so all PROCEDURE and FUNCTION declarations must precede any attempts to refer to them.*

Use of procedure identifier in expression

The compiler is processing an expression and finds the name of a procedure where the name of a variable, constant or function should appear.

RUN-TIME ERROR MESSAGES

G-Pascal has only five run-time error messages – these are explained below. A run-time error message is one that occurs while a program is running rather than compiling. When one of these messages occurs it will be followed by the words: 'Error occurred at P-code xxxx' where xxxx is the address of the P-code (instruction) currently being executed. If the meaning of the error is not immediately obvious, write down the P-code address and then recompile the program, asking for a listing during the compilation. By referring to the P- code addresses listed at the side of the compilation listing you can isolate which statement caused the error message.

Break ...

The program has been aborted by pressing the RUN/STOP key.

Divide by zero

An attempt has been made to divide by zero, which is not permitted. (The MOD operator, which is a form of divide, may also cause this error).

Illegal Instruction

This message should not appear in normal operation. It means that the P-code interpreter has encountered an invalid instruction (P- code). This could be caused by a program self-destructing somehow (perhaps by an array subscript going out of its allowable bounds) or invoking an 'independent' procedure which had not been loaded into memory at the correct address.

Illegal parameter in function call

One of the built-in graphics or sound effects procedures or functions has been called with a parameter (argument) outside its allowable range. For example, referring to sprite 9 in a SPRITE statement will cause this error, or voice 4 in a VOICE statement. Not all parameters are checked in this way – generally speaking the range check is carried out where an incorrect argument could have disastrous effects on the system if it went unchecked. In other cases the supplied value is truncated to fit into the required range – for example the filter frequency specified for the SOUND statement should be in the range 0 to 2047 – if the value exceeds 2047 then the supplied value modulus 2048 is taken.

Stack full

There is no more room on the stack for variable data. The stack consists of 3,790 bytes (this is the equivalent of 1,263 INTEGER variables or 3,790 CHAR variables). This could be caused by either declaring too many variables (e.g. an array of 2,000 integers), or by calling procedures or functions recursively too often. Each time a procedure or function is invoked 6 bytes are reserved on the stack for 'linkage' data (such as the return address – the address from which the procedure was called) plus any 'local' variables declared for that procedure or function.

FILE HANDLING

If you enter 'F' (for Files) from the Main Menu you will see:

```
(L)oad, (A)ppend, (P)rint, (D)os,
(S)ave, (N)oprint, (V)erify, (Q)uit,
(E)dit, (C)atalog, (O)bject?
```

This is called the 'Files Menu'. To choose one, press the letter corresponding to your choice (the letter in brackets). Do *not* press RETURN as well. To leave the Files Menu and return to the Main Menu press 'Q' (for Quit).

The choices are:

Load

This will load a G-Pascal source program from disk or cassette. *The loaded program will replace any currently in memory.* If you do not want to load anything just press RETURN. You will see:

```
(C)assette or (D)isk?
```

Press C for Cassette or D for Disk. Any other character will return you to the Files Menu.

You will then see:

```
File name?
```

Type in the file name of the file to be loaded (the program name, in other words) and press RETURN. When loading, appending or verifying from cassette the file name is optional – if any file will do just press RETURN. The program will now load.

Warning: There is *no* check to see if the program is too big. It is your responsibility to not load programs that are too big. This would not normally happen unless the file was created independently of G-Pascal. If you do load a file that is too big you will probably 'clobber' part of G-Pascal.

Append

This appends a program to the back of one that is currently in memory. If you do not want to append anything just press RETURN. Append is a powerful way of copying useful procedures or functions from one program to another. You could build a 'library' of useful procedures and functions and just Append them at the appropriate points in your program, thus saving a lot of typing and debugging effort. Apart from the fact that the Appended program goes at the end of the one currently in memory, Append works the same as Load so see the 'Load' command for what to type next. Be careful when appending that you do not append too much to fit into memory. G-Pascal does not check when appending that there is sufficient room for the new file.

Save

This saves your program on disk or cassette. *If saving to disk then the saved program will replace any of the same name on disk.* If you do not want to save your program just press RETURN. Apart from the fact that the program is being saved, not loaded, the Save command works the same as the Load command so see the 'Load' command for what to type next.

Verify

This verifies that the program recently saved on disk or cassette has saved properly. It reads the file on disk or cassette and compares it with the image in memory. Naturally for this to work properly it must be done before any changes are made to the program in memory. Apart from the fact that the file is being verified, not loaded, the Verify command works the same as the Load command so see the 'Load' command for what to type next.