PCQ_Pascal

The Amiga Compiler

COLLABORATORS

	TITLE : PCQ_Pascal	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	The Amiga Compiler	July 25, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

PCQ_Pascal

1.1 Introduction

-----=====>>>>> Pascal 'PCQ' 1.2b FreeWare <<<<<=====-----

By Patrick Quaid

^^

Contents - Main

Contents - PCQ 1.2

****

This guide was compiled by Mike Foreman from the Pascal.DOC included
in the PCQ Pascal 1.2 archive from the AmiNet.
This guide was compiled using the very helpful 'GMake' by Gary Greenhill.

1.2 Distribution Disclaimer

==========

Disclaimer.....Sort of.

==========

As there was no disclaimer in the PCQ archive I can only assume that
any errors or machine malfunctions that might occur by using PCQ is the
fault and responsibility of the user and not of Patrick Quaid.
With regards to this guide...
Any errors, machine malfunctions, detonation of nuclear devices, etc.,
are not the responsibility nor liability of Mike Foreman. * :-|

1.3 About the distribution

==========

About PCQ

==========

What have I got here?

What is included in the PCQ distribution depends on where you got
it from. If you got the disk from me, it most likely includes
everything. If you downloaded it, it probably does not include
several common files, nor does it include the source for the
compiler or runtime library. It does, however, contain some
combination of the following:
Pascal The compiler itself
Peep The peephole optimizer

Pascal.DOC This documentation file
Announcement Description of the registered version
Readme.PCQ A short description of PCQ
PCQ.lib The runtime library
Include.LZH The include file library, compressed with
the LHArc program.
Examples.LZH Several example programs, also compressed
with LHArc.
IDList.LZH A directory for the include files
PCQ.ced An Arexx program that lets you compile,
assemble, link and view errors all within
CygnusEd Professional.
A68k Charlie Gibbs' assembler (1)
A68k.doc The documentation for A68k (1)
A68k2do.txt Improvements planned for A68k (1)
History.log Version-by-version changes of A68k (1)
Blink The Software Distillery linker (1)
Blink.doc The documentation for Blink (1)
(1) These files are often omitted from the BBS archive version
of the distribution to cut download times. They are
already available on most systems, but if you are
uploading PCQ to a system, you might want to point out
that A68k and Blink are also necessary, and verify that
they are also available.

1.4 About the author

==========

Authors

==========

PCQ Pascal v1.2b - (Patrick Quaid)

I can be contacted about questions or whatever at:
Pat Quaid
2250 Clarendon Blvd, Apt #1209
Arlington, VA 22201
USA
Telephone: (703) 524-8945
You are much more likely to be able to contact me by mail than by

phone, but I certainly don't mind if you try. Enjoy the compiler.
PCQPascal.guide - (Mike Foreman)

This guide is made up of PASCAL.DOC included in the PCQ archive and
compiled by Gary Greenhill's Gmake utility.
How to contact me:
Until Summer 1997:

Address: Flat G2.8,
Brook Street Halls,
Brook Street,
High Wycombe,
Buckinghamshire,
England.
E-Mail: n6111388@buckscol.co.uk 5:-)
After that I shall update this guide to include contact info. :)
Other software by me: :-)
KillClick: Stop that drive click - (Amiga)
Dirv's Day: Graphic Adventure (M.I Style) - (Amiga)
Cookie2000: Futuristic fortunes - (Amiga)
!Switcher: Conversion program that converts almost ALL KNOWN
scales (Time, Weight, Distance, Speed, etc) - (Acorn)
!Cookie: A typical Amiga idea implemented on the Acorn. I have
never come across one of the on the Acorn - (Acorn)
!JukeBox: Multi-format module player - (Acorn)

1.5 How to install the files

==========
Installing PCQ
==========
How do I make a work disk?

Glad you asked. The answer is that I don't know. There are
jillions of different system configurations out there, so I can't
tell you exactly how to set your system up. But I can give you
some general guidelines.
First of all, somewhere in your command path you are going to need
the compiler, assembler, linker, and a text editor. Normally

these would go in the C: directory, but if you use the AmigaDOS PATH command, they can go in any of the path directories.

If you don't have a text editor, I heartily recommend CygnusEd Professional from ASDG. TxEd is supposed to be about as good, but I've never used it. Programmers on a budget can use MEMACS from the Extras disk, ED from the Workbench disk, or a variety of freely distributable editors like DME. ED isn't a great choice, MEMACS is fair, and from what I hear DME is really good. Take your pick.

The other file you'll definitely need is the runtime library, which can go anywhere. I keep mine in my work directory, but others like to keep it in their LIBS: directory. It's up to you, but don't forget where you put it.

These files are all absolutely necessary for even the simplest programs. If you want to use any Amiga routines, or use special routines from PCQ.lib, you will also need to install the include file library. This is where it gets a bit hairy, because the set of include files is just under 800 disk blocks (a little less than 400k). That's half a floppy disk right there. The size of these files is going to make it awfully hard to use the complete system on single drive systems. If you want to install the Include files you'll also need to find a copy of LHArc (or one of the compatible archivers). LHArc is available on virtually all bulletin boards, so you shouldn't have much of a problem finding a copy (if you have not yet done so, do yourself a favor and buy a modem. It's the best investment you'll make).

So how do we organize all this stuff? It depends on your system, of course, but here's a few examples. If you want to install the program on a hard drive, you could try the following. First, copy Pascal, A68k, and Blink to your C: directory (also copy a text editor if one is not already there). Then create a work directory. Copy PCQ.lib to this new directory. Now make a subdirectory of your work directory called something like "Include". It's actually not necessary that it's a subdirectory, but it will help you keep your disk structure straight. Then use the AmigaDOS ASSIGN command to make an assignment of Include: to your new include directory. Now you want to uncompress the include file library into your new include directory, with a command like:

LHArc -x -r x DistributionDisk:Include Include:
Use the actual path of the include archive, or course. Every time
you use the compiler, you should be sure to properly ASSIGN the
Include: directory - you'll probably want to put that assignment
in the startup-sequence or an initialization script.
If you have one disk drive, I recommend the following disk
structure. The problem here is that I don't have a one-drive
system on which to test this, so you may have to adjust it
somewhat.
Root:
PCQ.lib
C (dir)
A68k Assign
Avail Blink
CD Copy
Date Delete
Echo Ed
Else EndCLI
EndIf EndSkip
Execute Failat
If Info
Join Lab
List LoadWB
Makedir NewShell
Pascal Path
Prompt Protect
Quit Relabel
Rename Resident
Run SetPatch
Stack Type
Libs (dir)
diskfont.library icon.library
mathtrans.library version.library
Devs (dir)
ramdrive.device system-configuration
S (dir)
Startup-Sequence Make
L (dir)
Disk-Validator Newcon-Handler

Port-Handler Ram-Handler
Shell-Seg
Include (dir)
All the include files...
Trashcan (dir)

To make this sort of disk, first make a copy of a standard Workbench disk. Then prune away everything, and I mean everything, that you don't absolutely need. You'll note for example that the list above doesn't even have the serial.device and the parallel.device, so you can't use your modem or printer. These sorts of sacrifices will have to be made. Next, copy Pascal, A68k and Blink to the :c directory, then copy Make to the :s directory and PCQ.lib to the new disk's root. Make a new directory called Include, then CD to it. Uncompress the Include file directory by using the following command:

LHArc -x -r x Distribution:Include WorkDisk:Include/

Use the actual path for LHArc and the archive itself. You might have to run Install on the disk to make it bootable, but you'll have yourself a barely usable work disk for Pascal. You will have virtually no free space, so you'll have to edit and compile files in RAM:, copying them to a separate disk as often as possible. It will be a hard life. If you are only writing very simple, standard Pascal programs, which use none of the Amiga's special features, you can leave out the include directory (thus freeing up nearly 400k), but you won't be able to create very interesting programs. Single drive users probably have several tricks up their sleeves to get around disk space problems, so if you know of a better plan, by all means use it.

If you have two disks, I would recommend a setup similar to the single drive system. Move the include directory to the second disk, however, and flesh out the system files on the boot disk (so you can use your printer, especially). Use the ~400k of free space on your second drive as your work area. This is the sort of setup I used when I originally developed PCQ.

If you have lots of RAM (say, a megabyte or more), you might want to copy some of this stuff into RAM: or RAD: to free up work space. For example, if you have one megabyte of RAM you might be able to copy the entire Include directory to the RAM: disk and still have enough memory to compile medium sized programs.

The one optional part of the system is the peephole optimizer, Peep. This program takes the assembly output of the compiler and makes it somewhat more efficient and slightly smaller. If you are compiling on floppy disks you'll probably want to skip it, but if you are working in RAM or on a hard disk it's doesn't add much to the compilation time. Peep should be copied to the same place as Pascal.
However you end up configuring your system, you'll have to define the Include: assignment to point to your include file directory. This assignment is just the convention I used in the include files and example programs - it's not part of the compiler itself, in other words - so if you are willing to modify all the include files you can use whatever assignment you like. The only other assignment of interest is the T: directory. This one is automatically created by AmigaDOS, and since it is used by the Make script to store intermediate files, assigning it to a RAM: directory speeds up the whole compilation cycle dramatically.

1.6 How to use the package

==========

Compiling a Program

==========

Turning a Pascal source file into an executable program is (at last count) a four step process. You start off with your source code (one of the example programs, for example), which probably ends with ".p". The first step is to use the compiler to turn this source file into an equivalent assembly language program. To do this, invoke the compiler with the following command format:
Pascal InputPascal OutputAssembly [-q] [-s] [-b]
"InputPascal" should be replaced with the complete filename (including path, if necessary) of your source file, and "OutputAssembly" is the filename (optionally with a path) of the assembly language output file. I normally use the extension ".asm" on the output file. The command line arguments can be placed anywhere on the command line, in upper or lower case, and have the following meaning:
-q Run in "quiet" mode, i.e. the compiler does not write anything to the screen except error

messages. This makes it easier to parse the
output of the compiler in an ARexx script, for
example.
-s Small initialization. Small programs use a
different initialization routine that does not
have all the overhead of the standard routine,
but it means that you will not be able to use any
Pascal IO (e.g Writeln(), Readln(), etc).
-b Disable short-circuit Boolean expressions.
Normally, PCQ abandons evaluation of a Boolean
expression as soon as its overall value becomes
known (if the left side of an AND expression is
false, for example, there is no point in
evaluating the right side). Thus some parts of
the program will not be executed. This option
forces PCQ to fully evaluate all Boolean
expressions.
The second step takes the assembly language file and creates a new file that's slightly more efficient. This is done using the program Peep, which is invoked with the following command format:

Peep InputAssembly OutputAssembly

I normally use the extension ".s" for assembly files that have been run through the peephole optimizer, but it is entirely up to you.

The third step takes the .asm or .s file and creates an object file. An object file is mostly the same machine-readable instructions and data as the executable file, but it does not contain all the routines the final program will need. The assembler, in this case Charlie Gibbs' A68k, converts assembly language programs to object files, and is invoked with the following command line:

A68k InputAssembly OutputObject

A68k is a very powerful, flexible program with lots of options - be sure to read its documentation file for a complete description.

If all these steps finished without errors, you can link the object file to all the routines in the runtime library it needs to be an executable program. The Software Distillery linker, Blink, handles this part of the process. It is invoked with the following command line:

Blink InputObject to OutputExec library PCQ.lib
Blink also has many options, and comes with a documentation file that completely explains them. Note that you might have to use a full path to specify where PCQ.lib is - if it's in the LIBS: directory, for example, you would use LIBS:PCQ.lib instead.
You now have, finally, a finished executable file. That seems like a lot of steps, and a lot to remember, just to make one program. It is. Therefore I use, and have included in the distribution, a couple of AmigaDOS scripts that automate the whole process. The first is called Make, and it compiles a program in the T: directory, leaving just the executable program on the disk. It looks like this:
.key source
Pascal <source>.p T:<source>.asm
A68k T:<source>.asm T:<source>.o
Delete T:<source>.asm
Blink T:<source>.o to <source> library PCQ.lib
Delete T:<source>.o
You'll note that my preferences for extensions are built in to this script - the Pascal file ends in .p, the assembly language file ends in .asm, and the object file ends in .o. If you prefer other names, just change the script. Also note that you might have to change the script if your compiler, assembler and linker are not on the normal path, or if PCQ.lib is not in the current directory (in both cases, just use the complete path in place of the file name). To run this script, you just invoke it like this:
Make ProgramName
Note that you leave off any extensions. If everything works OK, you'll have an executable program called ProgramName left in the current directory. One problem with this script is that you can't use any command line options on the compiler. I actually have several different versions of Make scripts that handle all the combinations I use.
The other script, OMake, is virtually the same, except it also invokes the peephole optimizer to make slightly more efficient programs. It looks like:
.key source
Pascal <source>.p T:<source>.asm
Peep T:<source>.asm T:<source>.s

Delete T:<source>.asm

A68k T:<source>.s T:<source>.o

Delete T:<source>.s

Blink T:<source>.o to <source> library PCQ.lib

Delete T:<source>.o

Again, you might have to make modifications to suit your set up.
If you are working on a fairly complex project, you will probably
find it convenient to make a set of specialized scripts to
automate the compile-assemble-link process.

1.7 Incompatibilities

==========

Incompatibilities

==========

PCQ Pascal is incompatible with standard Pascal, and by
extension Turbo Pascal, in several ways. Briefly, they are:
(1) Sets are not supported.
(2) The standard Pascal declaration of pointers to records is
not supported. For example, the following syntax is legal
in standard Pascal:

```
type
WindowPtr = ^Window;
Window = record
NextWindow : WindowPtr;
...
```

That's an exception in standard Pascal - the only time you
can use an identifier before you declare it. In PCQ
Pascal, that syntax will fail with an Unknown ID error.
In its place, use something of this form:

```
type
Window = record
NextWindow : ^Window;
....
end;
WindowPtr = ^Window;
```

(3) Variant records are not supported.
(4) The familiar syntax for specifying a single quote
character constant, which in standard Pascal looks like

"", is not supported. Instead PCQ Pascal uses C escape
conventions, which are explained in the section called
Strings.
(5) The way you open a file is different from Standard Pascal,
although once it's open the commands are basically the
same.
Although PCQ Pascal was not designed to be compatible with Turbo
Pascal, in order to make porting programs easier I'll point out a
few of the more important differences (in addition to those
above).
(1) PCQ Pascal handles strings completely differently from
Turbo Pascal. PCQ strings are similar to C strings, which
are not as easy to manipulate as Turbo strings. In fact,
PCQ strings are the most difficult of the three. They are
fully explained in the Strings section.
(2) In a PCQ Pascal function, assigning a value to the
function name causes you to leave the function. In Turbo
Pascal, the function name is treated as a write-only
variable, and you have to use the Exit command explicitly.
All the examples in the Pascal Report skirt the issue by
assigning the value as the last statement.
(3) Speaking of Exit, Exit in PCQ Pascal quits the program,
whereas in Turbo it quits the current function or
procedure. Turbo's "Exit" is like PCQ's "return", and
Turbo's "Halt" is like PCQ's "Exit". Got that?

1.8 PCQ Programs, reserved identifiers & structures

==========

PCQ Programs

==========

For the most part, PCQ Pascal programs look very much like
standard Pascal programs. Apart from the incompatibilities
described above, in fact, PCQ should compile plain Pascal programs
directly. PCQ makes several extensions to Pascal, however, so in
order to explain them I'll go over what is and is not allowed in
PCQ programs.

==========

Reserved Words

==========

Reserved words are symbols that cannot be used as identifiers in
your program. They have special meaning to the compiler, and
cannot be overridden. The reserved words of PCQ are as follows:

and for procedure
array forward program
begin function record
by goto repeat
case if return
const in set
div label then
do mod to
downto not type
else of until
end or var
external packed while
file private with

As you can see, even the unimplemented stuff is reserved.

==========

Pascal Program Structure

==========

The normal structure of Pascal programs is as follows:

<Program> ::= Program <Identifier> ; <Block>. |
Program <Identifier> (<Identifier List>); <Block>.
<Block> ::= <Declarations> begin <Statements> end |
begin <Statements> end
<Identifier List> ::= <Identifier> , <Identifier List> |
<Identifier>

Does everyone read Backus-Naur? The idea is that the objects to
the left of the definition sign ::= are being defined by the rule
to the right. Objects in angle brackets are defined elsewhere in
the list, and the vertical bar character separates alternatives.
Everything else is to be taken literally. Thus a Pascal program
begins with the literal word Program, followed by an identifier,
followed by a semicolon, followed by a block (whatever that is),
and ending with a period. On the other hand, it can also consist
of Program followed by an identifier, followed by a left
parenthesis, followed by an identifier list, followed by a right
parenthesis, a semicolon, a block, and finally a period. Once you

get used to them, BNF diagrams will tell you exactly what you need to know about the syntax of a program. BNF grammars can also be represented as "railroad diagrams", and if this weren't a text-only document I'd be using them instead.

We've left a few things undefined, so we better start filling it out.

<Declarations> ::= <Declaration> , <Declarations> | <Declaration>

<Declaration> ::= <Type Declaration> |

<Constant Declaration> |

<Variable Declaration> |

<Label Declaration> |

<Procedure Declaration> |

<Function Declaration>

Note that standard Pascal imposes an order on the declarations: it says there should be one constant block, followed by one type block, etc. PCQ Pascal removes that restriction, so you can have as many declarations in whatever order you see fit.

Subject to the definition of the rest of those objects, this is the format for any normal PCQ Pascal program. There is an exception (of course): the separately compilable file. Turbo Pascal provides a Unit structure so you can write, compile and debug units, then use them in many programs. PCQ provides a similar, although not as powerful, feature in the form of external files. Consider the following extension of our basic rule:

<Program> ::= Program <Identifier>; <Block>. |

Program <Identifier> (<Identifier List>); <Block>. |

External ; <Declarations>

According to this, an external file is just the reserved word External followed by a semicolon, then a series of procedures, functions, whatever. See the section called External Files for more information.

1.9 Type declarations

==========

Type Declarations

==========

Type declarations take the following form:

<Type Declaration> ::= Type <Type Definitions>;

<Type Definitions> ::= <Type Definition>; <Type Definitions> |
<Type Definition>
<Type Definition> ::= <Identifier> = <Type Specification>
<Type Specification> ::= <Identifier> |
<Range> |
^<Type Specification> |
(<Identifier List>) |
<Array Definition> |
<Record Definition> |
file of <Type Specification>
<Range> ::= <Constant Expression> .. <Constant Expression>
<Array Definition> ::= array [<Range>] of <Type Specification> |
array <Identifier> of <Type Specification>
<Record Definition> ::= record <Variable Definitions> end
Once you digest the BNF, I think you'll find that straight-
forward. Constant expressions are discussed in the Expressions
section below - for now let's look at the types that are
predefined in PCQ Pascal.
Numeric Types

PCQ Pascal supports a variety of numeric types that should give
you the flexibility you need to carry out whatever math you might
require. The types, from smallest to largest range, are:
Byte This is a 1-byte, unsigned integer, capable of holding
a value from 0 to 255.
Short A Short is a 2 byte (16 bit) signed integer, with a range
of -32768 to 32767.
Integer Integer is the largest ordinal type. It holds a 32-bit
signed integer, with a range of about -2 billion to
2 billion.
Real A floating point value, in Motorola Fast Floating Point
format. A Real value is 4 bytes (32 bits), and can
express floating point values from about 10 x 10^18 to
about 5 x 10^-20, positive or negative. I'm not sure
what the accuracy of FFP is, but I wouldn't count on
more than 5 or 6 digits.
The numeric types are interchangeable in expressions. If you use
two different types in, for example, a plus expression, the
compiler will automatically promote both values to the smallest

"container type". A container type is a numeric type whose range covers both of the arguments. For example, the result of adding a Short to an Integer is an Integer. The result of most binary operations involving Real values will be a real value.

Other Ordinal Types

Integers, Shorts and Bytes are all Ordinal types. This means they are discrete values that can be represented exactly in binary (unlike floating point values, which are approximated). The other ordinal types are:

Char A one-byte value, stored as the ASCII value.

Boolean Also one byte long, a Boolean value is either -1 (all binary ones) for true, or 0 for false. Anything else could produce unpredictable results. Incidentally, the Boolean type can be considered an enumerated type with the values True and False. In most implementations False < True and Succ(False) = True, but not in PCQ.

Enumerated The values of an enumerated type are specified within the program. If there are 256 or fewer values, the enumerated type will be stored in one byte. If there are more, it will be stored in two bytes. In either case, each enumeration is associated with an integer value, starting with 0.

Pointer Types

Pointer types hold the address of another variable. They are all 32-bit values. The predefined pointer types are:

Address Address is a special pointer type that is compatible with all other pointer types. It should not be dereferenced, but you can use it to avoid cumbersome type casts.

String The String type in PCQ is actually defined as a pointer to char, although it has other special qualities like allowing array-like subscripts. Strings are explained in the section called Strings.

File Types

Files in PCQ are based on normal AmigaDOS files, but provide

automatic buffering and access to the handy Pascal IO routines
(Writeln, et al). File variables come in two varieties:
Text A Text file is any normal ASCII file (this
documentation, for example.
Typed Files A typed file, declared as "File of Type", only stores
values of the given type. If you issue a Write()
command on a typed file, PCQ will output the actual
binary representation of the object to the file. For
example, if you have a "File of Integer", the typed
file will store each Integer as a four-byte binary
representation. A text file, on the other hand, will
store each Integer as a series of ASCII characters
from '0' to '9'. You cannot use Writeln and Readln on
typed files.
Type Compatibility

There are two kinds of type checking. One is for normal type
compatibility. That's used for comparing arguments in an
expression, or for formal and actual value parameters. The second
kind checks for identical types, and is used for more strict
circumstances: in assignment operations and between formal and
actual parameters passed by reference (using the VAR keyword).
PCQ Pascal is, relative to Standard Pascal, generous in its
enforcement of these rules. For simple type compatibility, the two
types have to pass on of the following tests:
o They are the same type, or
o They are both number types (real, integer, etc), or
o They are both arrays with the same size range and the
element types are compatible, or
o They are both pointers to compatible types, or
o They are both files of compatible types.
Synonym types (e.g. "TYPE ByteSynonym = Byte") are considered
identical types. For the more strict type identity check, remove
the second rule.

1.10 Constant declarations

==========

Constant Declarations

==========

Constants are normally declared in the Standard Pascal fashion,
which looks like the following in BNF:
<Const Block> ::= Const <Const Declarations>;
<Const Declarations> ::= <Const Declaration> |
<Const Declaration> ; <Const Declarations>
<Const Declaration> ::= <Identifier> = <Constant Expression> |
<Identifier> : <Type Specification> =
<Typed Constant Value>
<Typed Constant Value> ::= <Constant Expression> |
(<Constant Expressions>) |
@ <Identifier>
Constant expressions will be defined below, but you can think of
them as any normal expression that can be completely evaluated
during the compilation (i.e. they use no user-defined functions
and access no variables).
Normal constants are easy to understand - they are just like
read-only variables, but they actually take up no memory. The
constant value is just inserted in the program instead of the
identifier. Since they are formed by constant expressions, normal
constants can only have a few different types: integer, real,
char, Boolean, string and array of char. They can't have any
user-defined type, and can't be record or array types.
Typed Constants

That's where typed constants come in. Typed constants are best
thought of as pre-initialized variables, and they can be of any
type you can define, besides file types. I have no idea why Turbo
Pascal decided that these objects should be called constants
rather than variables, but I'm following their lead. Your program
loads with these values already in place, and they will not be
refreshed until the program is reloaded. They will, therefore,
screw up resident programs if used incorrectly.
One handy side effect is that variables local to some procedure
can retain their value throughout a program's execution (like

static variables in C, right?). In other words, if you declare a typed constant within a procedure, it will not be accessible to any code outside of that scope, and it will not lose its value between calls to the procedure. Looks like a local, acts like a global.

Declaring typed constants is error prone, and you invariably spend most of the time counting a list of numbers or trying to match parentheses or something equally mind-numbing. Nonetheless, they are awfully handy. The syntax corresponds to the second alternative of the <Const Declaration>, but that doesn't tell the whole story. Perhaps it's best to look at some examples:

```
Type
ExampleRec = record
Field1 : Integer;
Field2 : Char;
Field3 : Boolean;
Field4 : ^Integer;
Field5 : Array [-1..1] of Byte;
end;
MultiDim = Array [0..1,0..1] of Integer;
Const
Message1 : String = "An example string";
Message2 : Array [0..9] of Char = 'String2 ';
Message3 : Array [0..9] of Char = ('a','b','c','d','e',
'f','g','h','i','j');
Value1 : Integer = 456;
Record1 : ExampleRec =
(34, 'r', True, @Value1, (34,56,12));
Multi1 : MultiDim = ((34,12), (45,15));
```

Note that all the "Message" constants and Value1 could also be specified as normal constants. Message1 and Value1, in that case, would not take up any memory, but Messages 2 and 3 would be the same. The format of the Message3 declaration is typical for arrays, but the Message2 format is a special abbreviation for arrays of char.

There are several differences between this and Turbo Pascal's syntax (all, if I may say so, in favor of PCQ). First of all, definitions of constant records in Turbo Pascal requires that you use the field identifier, followed by a colon, the value, and

possibly a semicolon. That’s far too cumbersome for me, so I just use the same idea as arrays: you specify each field, separated by commas, in order.

The second difference with Turbo Pascal is that, until version 6.0, it only allowed pointer types to be initialized to Nil. PCQ Pascal allows you to initialize a pointer type as Nil, or as the address of a preceding global variable or typed constant. To do this, you normally use the ’@’ operator, which returns the address of a specified variable.

The last difference is that you can use some typed constant values in subsequent constant expressions. You can’t use any structured types (i.e. records and arrays), but you can use integers, reals, and other so-called "simple" types.

Standard Constants

There are several standard constants (i.e. they are built in to the compiler itself - you don’t have to declare them or use an include file). They are:

False False is an enumeration of the type Boolean, and has the value 0.

MaxInt MaxInt is the largest integer that can be stored in the 32-bit Integer type. It is: 2,147,483,647, which is $7FFFFFFF in hex. Thus you can say that the range of an Integer variable is +MaxInt to -MaxInt.

MaxShort MaxShort is the largest number that can be stored in the 16-bit Short type. It turns out to be 32,767, or $7FFF in hex. Just as with MaxInt, you can say that the range of a Short variable is +MaxShort to -MaxShort (to be precise, the range is actually +MaxShort to -(MaxShort+1), and MaxInt is analogous).

Nil Nil is a constant of type Address. It is defined as something like:

Nil = Address(0);

In Standard Pascal, "Nil" is a reserved word. In PCQ Pascal it isn’t reserved, for the simple reason that it doesn’t have to be.

True True is an enumeration of the type Boolean, and actually has the value -1. In some cases, any non-zero number will suffice, but the "not" operator will behave erratically if you use weird values.

1.11 Variable declarations

==========

Variable Declarations

==========

Variable declarations are handled just like standard Pascal:
<Var Declaration> ::= Var <Var Definitions> ;
<Var Definitions> ::= <Var Definition> |
<Var Definition> ; <Var Definitions>
<Var Definition> ::= <Identifier List> : <Type Specification>
Global variables, i.e. variables declared at the outer-most level,
are allocated as static memory. All variables declared within a
procedure or function are allocated on the stack, so if you are
writing re-entrant routines, be sure to avoid writing to global
variables.
Variables larger than one byte are always allocated on word
boundaries (not longword boundaries - you'll have to use AllocMem
to guarantee that). That's an option in Turbo Pascal, but the
68000 makes it mandatory for Amiga programs. Note that this
applies within records and arrays as well, so be careful if you
make assumptions about variable size and location.
Keep in mind that local variables (not typed constants) disappear
when you leave the function, so you shouldn't try to access them
afterward (with a pointer variable, for example). It just won't
work.
Standard Variables

There are several variables built-in to PCQ Pascal. They are
always available, and are treated like global variables. They
are:
CommandLine This is a String variable that points to the
command line the user entered (with the name of
the program and any indirection parameters
stripped off). If the program was executed from
the Workbench, this value is not defined. Note
that, to be on the safe side, this variable should
be treated as read-only. Make a copy of it if you
want to modify it.
ExitAddr ExitAddr is the location within your program that

a runtime error occurred. If no error occurred,
its value is undefined. See the section called
"Exit Procedures" for more information.
ExitCode This is the Integer value that will be returned to
AmigaDOS when your program terminates. It is only
defined within an exit procedure.
ExitProc This is an Address variable that holds the address
of the first procedure that will be executed when
your program terminates. See the section called
"Exit Procedures" for more information.
HeapError HeapError is an Address variable that holds the
address of a function to call when New() or
AllocString() is unable to allocate a block of
memory. See the section called Memory Management
for more information.
Input Input is a Text file type. It corresponds to the
standard input channel of the program, such as the
CLI from which it was run. PCQ Pascal programs
always establish some sort of standard Input file
unless you specifically tell it not to. See the
section called Input/Output for more information.
Output Output is a Text file type. It corresponds to the
standard output channel of the program, which in
most cases is the CLI from which it was run. PCQ
Pascal programs always establish an output file
unless instructed not to.

1.12 Procedure and Function declarations

==========

Procedure and Function Declarations

==========

Procedures and functions allow you to define common routines
within a larger block. Procedures are executed by procedure
statements, and function are executed when they are used in
expressions. The format for procedure and function definitions is
as follows:
<Proc Declaration> ::= Procedure <Identifier> ; <Body> |
Procedure <Identifier>

(<Formal Params>); <Body>
<Body> ::= <Block> | External | Forward
<Formal Params> ::= <Identifier List> : <Type Specification> |
var <Identifier List> : <Type Specification>
<Func Declaration> ::= Function <Identifier> :
<Type Specification> ; <Body> |
Function <Identifier> (<Formal Params>) :
<Type Specification> ; <Body>
At the risk of being repetitive, keep in mind that you can only use procedures where the BNF calls for a <Statement>, and you can only use functions where the BNF calls for an <Expression>.

Parameter Passing

Pascal provides two different kinds of parameter passing. The first type are called value parameters. These are the normal case. When you pass a parameter by value, the compiler makes a copy of the value, and sends that along. It's as if someone tells you to study the Mona Lisa, providing some photographs for you to look at. You can draw a moustache on the photograph, but it won't affect the Mona Lisa, and whatever the procedure or function does to the parameter will not affect your original values. When you are calling the procedure, you can use any expression (of the correct type, of course) as the actual parameter.

The other type of parameters are called reference parameters. They are specified by preceding the parameter name with the reserved word "var" in the procedure or function header. If you use reference parameters, it's as if someone tells you to study the Mona Lisa, sends you to the Louvre, and tells you where to find the painting. Thus you are working on the original, and had better be careful about it. Reference parameters must be given as variable references - they cannot be full expressions. Also, reference parameters have to pass a stricter type compatibility test than value parameters do, since we can't have the procedure trying to write a 4-byte Integer into a 1-byte space.

Normally you should prefer value parameters over reference parameters (it tends to cut down on bugs), except in three cases. The first is when you actually do want to affect the value of a variable. The second is when the parameter is large. If you use a value parameter, the compiler must make a complete copy of it on

the stack before it calls the routine, which uses both time and stack space. If you use a reference parameter, the compiler just puts the address on the stack. The third case is when the parameter is a file type, in which case it must be passed by reference.

PCQ Pascal pushes arguments on the stack from left-to-right. C compilers, because they have to support variable numbers of parameters, push arguments from right-to-left. Therefore if you plan to call C routines, you will have to re-order either the order in the call or the order in the routine.

Forward Procedures and Functions

In Pascal, every identifier must be declared before it can be used. This creates problems when procedures or functions are mutually dependent, so Pascal allows you to pre-define procedures and functions that will be fully defined farther down in the source code.

A forward reference looks just like a normal procedure or function declaration, but has the reserved word "Forward" right after the header. The header contains all the information the compiler needs to correctly call a routine, so after the forward declaration the routine can be used. When the routine is eventually defined, the compiler ensures that it matches up with the definition given earlier. This means that there must be the same number of parameters, the parameter types must be the same, and if it’s a function, the result type must be the same. Note that the actual names of the parameters can be different - only their number, order and type are compared.

A forward reference must be resolved within the same block as it was declared, and the same routine cannot be forward-referenced more than once.

External References

External references are very much like forward references - they are declarations of routines that aren’t actually fully defined. The difference is that forward references must be defined later on in the same program file, whereas external references are not defined in the source file at all.

The declaration of external references is the same as that of

forward references, but the reserved word "External" is used in place of "Forward".

External references are used for two main purposes. First of all, they allow you to use procedures and functions defined in External files. All the system routines declared in the include files, for example, are actually defined in External files. They also allow you to call routines compiled in a different language, as long as you follow some guidelines in parameter passing. See the section called External Files for more information on using external routines.

Standard Procedures

There are several procedures built-in to PCQ Pascal. They include:

Close(FileVariable : Any file type)

The "Close" procedure disassociates the FileVariable from the physical disk file, and frees up all the system resources used to keep the file open. See the Input/Output section for more information.

Dec(V : Any ordinal or pointer) or Dec(V, Amount : Integer)

Dec (which is short for decrement) subtracts one (or more) from a variable. "Dec(x,n)" is the same as "x := x-n", but slightly more efficient. The variable must be either an ordinal type, like Char, Byte, Integer, etc., or a pointer type. The amount parameter, which is optional, must be an integer type - if it is not included, the value one is assumed.

If the variable is an ordinal type, the value is subtracted normally. If the variable is a pointer type, however, Dec() multiplies the amount by the size of the type to which the pointer refers. In other words, if you have a pointer p of type ^Integer, the statement Dec(p) actually subtracts 4 from the pointer.

Dispose(Variable : ^Anything)

The "Dispose" procedure frees memory previously allocated with "New" (see below). See the section called Memory Management for information on PCQ's memory routines.

Exit or Exit(ReturnCode : Integer)

Exit, like the AmigaDOS and the C function of the same name, terminates the program. If a return code is specified, it is returned to AmigaDOS. By convention, return codes are normally 5

for small problems (warnings), 10 for significant problems (errors), and 20 when everything goes wrong (failure). Using Exit without a return code is the same as Exit(0), which indicates that everything is O.K.

Exit is the system-safe way to quit a program - you should not call AmigaDOS's DOSExit() routine. When you call Exit, all the exit procedures are called in turn. This includes the standard routine that closes all open files and deallocates memory, as well as any that you define.

Get(var FileVariable : A file type)

The "Get" procedure moves the file pointer to the next element in the file, without actually reading any element in. The element in the file buffer can be accessed through the FileVariableˆ syntax. See the Input/Output section for details.

Inc(VarReference: Ordinal or Pointer type) or
Inc(VarReference : Ordinal or Pointer, Amount : Integer)

The "Inc" command adds an integer Amount to the variable VarReference. If no amount is specified, it is assumed to be 1. If the variable is an ordinal type the addition is carried out normally, but if it is a pointer type the amount is first multiplied by the size of the object to which the pointer refers. In other words, Inc(p), where p is a pointer to Integer, actually adds 4 (the size of an Integer in bytes) to p.

New(PointerVar : ˆAnything)

The "New" procedure allocates a block of memory equal in size to the type to which the variable (which must be a pointer type) points. The address of the newly allocated block will be assigned to the variable. See the section called Memory Management for more information.

Put(var FileVariable : A file type)

The "Put" routine advances the file pointer past the current element, writing the current element to disk if necessary. The current element can be set through the FileVariableˆ syntax. See the section called Input/Output for details.

Read(var FileVariable : Text or File, Variable References....)

The "Read" procedure inputs information from an AmigaDOS file. The FileVariable is optional, and the procedure is one of the very few in Pascal that can take a variable number of arguments. See the Input/Output section for more information.

Readln(var FileVariable : Text, Variable References....)
The "Readln" procedure is very similar to "Read", but it can only be used on Text files, and after it has read all the arguments it requires, it continues to eat characters until it is at the beginning of the next line. See the Input/Output section for more information.
Reset(var FileVar : File or Text;
FileName : String;
BufferSize : Integer);
The "Reset" procedure opens a file for reading. The BufferSize parameter is optional. If you have used the {$I-} directive to indicate manual IO checking, you should be sure to check IOResult to see if the file opened correctly. If you are using automatic IO checking, the program will terminate with a runtime error if the file does not open correctly. See the Input/Output section for more information.
Rewrite(var FileVar : File or Text;
FileName : String;
BufferSize : Integer);
The "Rewrite" procedure opens a file for output, erasing any existing file of the same name. The BufferSize argument is optional. If the file cannot be opened, IOResult will be set to a non-zero value. If you are using automatic IO checking, that will cause your program to terminate with a runtime error. If you are using manual IO checking, be sure to look at IOResult to be sure the file is open. See the Input/Output section for more information.
Trap(TrapNum : Integer)
The "Trap" procedure can be used by some debuggers as a sort of automatic breakpoint. If you insert the correct trap, your debugger might stop at that location. Then again it might not.
Write(FileVariable : Text or File, Expressions....)
The "Write" procedure outputs information to an AmigaDOS file. The FileVariable is optional, and it can take a variable number of arguments. See the Input/Output section for details.
Writeln(FileVariable : Text, Expressions....)
The "Writeln" procedure is very similar to "Write", but it only works on Text files, and it terminates the line after writing all of its arguments. Again, see the Input/Output section for more

information.

Standard Functions

There are also a bunch of functions built in to the Pascal compiler. They are:

Adr(Variable Reference) : Address

The Adr() function returns the actual address in memory of the parameter. This is equivalent to Turbo Pascal's Addr() function, and as in Turbo Pascal you can also use the @ operator.

Abs(numeric expression) : same numeric type

The function call Abs(n) returns n, if n is positive, and -n if n is negative.

ArcTan(numeric Expression) : Real radians

The ArcTan() function returns the approximate arctangent of the parameter in radians.

Bit(BitNumber) : Mask Integer

The Bit() function returns an Integer with just the specified bit set. It is equivalent to (1 shl BitNumber).

Chr(numeric expression) : Char

The Chr() function transforms any numeric type into its ASCII character equivalent.

Ceil(Real expression) : Real

Ceil() returns the least integer greater than or equal to the parameter.

Cos(numeric expression in radians) : Real

The Cos() function returns the cosine of the given angle measured in radians.

EOF(file or Text variable) : Boolean

EOF returns True if the file is at the end-of-file position, or False otherwise. The EOF function is only valid for files opened for input. See the section called Input/Output for more information.

Exp(numeric expression) : Real

The function Exp(x) returns e raised to the xth power. In case it slipped your mind, e is about 2.71828....

Float(Integer expression) : Real

The Float() function converts any Integer type expression to its floating point representation. If the integer is fairly large, it might be approximated. Thus "Trunc(Float(IntVar)) = IntVar" is

not always true.

Floor(Real expression) : Real

The Floor() function returns the largest whole number less than or equal to the parameter.

IOResult : Integer

The IOResult() function checks to see if any errors have occurred since the last time you checked. Normally, PCQ Pascal inserts statements to check the IOResult automatically, but if you have set IO checking off (using the {$I-} directive), you'll have to check it explicitly. The act of checking it erases it, so if you need to re-use it you'll have to save it in a variable. See the Input/Output section for more information.

Ln(numeric expression) : Real

The Ln() function returns the natural logarithm (i.e. the logarithm to the base e) of the parameter.

Odd(numeric expression) : Boolean

The Odd() function returns TRUE if its argument is odd, and FALSE otherwise.

Open(fname : String;
var fvar : file or Text;
BufferSize : Integer) : Boolean;

Open is the function form of the Rewrite procedure. It opens a file for output, erasing any existing file with the same name. If the file opens correctly, Open returns True. If not, it returns False. The BufferSize parameter is optional. See the section called Input/Output for more information.

Ord(Ordinal expression) : Integer

The Ord() function returns the ordinal value of the argument as an Integer. In other words, it can convert Chars, Booleans, and enumerated types into their numeric equivalents. Since it actually changes the type of the argument into an Integer it is referred to as a transfer function.

Pred(Ordinal expression) : same type

The Pred() function returns the next least ordinal value in the same type. For example, Pred(2) is 1. If the argument is the lowest value within the type (such as Pred(ByteVar) where ByteVar is zero), its behavior is undefined.

ReOpen(fname : String;
var fvar : file or text;

buffersize : Integer) : Boolean
ReOpen is the function form of Reset. It opens a file for reading, and returns True if everything went O.K. If for any reason the file did not open correctly, ReOpen returns False. The BufferSize parameter is optional, and can be any amount. See the section called Input/Output for more information.
Round(Real expression) : Integer
The Round() function rounds a real expression to the nearest Integer. It is actually implemented as Trunc(value + 0.5), so it takes somewhat longer to execute than the Trunc function. Also, it always rounds 0.5 up.
Sin(numeric expression in radians) : Real
The Sin() function computes the sine for the given angle measured in radians.
SizeOf(Type Identifier) : Integer
The SizeOf() function returns the actual size of the given type. Note that the type must be specified as a single identifier.
Sqr(numeric expression) : same type
The function Sqr(x) returns x*x.
Sqrt(numeric expression) : Real
The function Sqrt(x) returns the square root of x. In other words, Sqrt(x) * Sqrt(x) = x.
Succ(Ordinal expression) : same type
The Succ() function returns the next greater ordinal value within the same type. If that value is not defined (for example, Succ(n) where n is the largest enumeration within a type), the function is not defined.
Tan(numeric expression in radians) : Real
Returns the tangent of the angle measured in radians. If the parameter is an odd multiple of Pi/2, this function will take on meaningless values (the tangent function is undefined at odd multiples of Pi/2).
Trunc(real expression) : Integer
The Trunc() function returns the integer portion of a floating point number. It is the fastest way to convert a real value to an Integer value.

1.13 Label declarations

==========

Label Declarations

==========

Labels must be declared in the declaration section before they can
be used in the program. The label declaration section has the
following form:
<Label Declarations> ::= Label <Identifiers> ;
<Identifiers> ::= <Identifier> | <Identifier> , <Identifiers>
Note that Standard and Turbo Pascal also allow labels to have
numeric values, but PCQ Pascal only allows normal identifiers to
be used. If you run across a program with numeric labels, you can
just put some letter on the front to make them legal identifiers.
Perhaps a better solution would be to trash the program.

1.14 Expressions

==========

Expressions

==========

PCQ Pascal accepts expressions very similar in form to just about
all Pascal implementations. It's probably very familiar to you,
but just to be on the safe side, we'll define it. Let's start off
with the constants accepted by PCQ:
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<digits> ::= <digit> | <digit> <digits>
<binary digit> ::= 0 | 1
<binary digits> ::= <binary digit> |
<binary digit> <binary digits>
<hex digit> ::= <digit> | a | b | c | d | e | f
<hex digits> ::= <hex digit> | <hex digit> <hex digits>
<Float Const> ::= <digits> . <digits>
<Integer Const> ::= <digits> | $ <hex digits> | % <binary digits>
<Char Value> ::= <ASCII char> | \ <escape sequence>
<Char Values> ::= <Char Value> | <Char Value> <Char Values>
<Number Const> ::= <Integer Const> | <Float Const>
<Char Const> ::= ' <Char Value> '
<String Const> ::= " <Char Values> "

<Char Array Const> ::= ’ <Char Values> ’

That should confuse things a bit. What it says is that whenever the compiler expects an integer, you can use decimal, hexadecimal or binary representation. It also says that floating point constants are only recognized in the normal 23.23 form, without any trailing exponent. Finally, it says that you can use C-style escape sequences for any text constant - these escape values are discussed in the Strings section.

So how are these things grouped into complete expressions? Consider the following definitions:

<expression> ::= <simple> <rel op> <simple> | <simple>

<rel op> ::= = | <> | < | > | <= | >=

<simple> ::= <term> <add op> <term> | <term>

<add op> ::= + | - | or | xor

<term> ::= <factor> <mul op> <factor> | <factor>

<mul op> ::= * | / | div | mod | and | shr | shl

<factor> ::= <Number Const> |

<Sign> <Number Const> |

<Char Const> |

<String Const> |

<Char Array Const> |

<Variable Reference> |

@ <Variable Reference> |

not <factor> |

(<Expression>) |

<Function call>

What does all this mean? It means that expressions are built up from factors, the basic blocks of an expression. A factor can be a constant, a variable, a parenthesized expression, and all the other things listed. The following examples are all factors:

23

(4.0 - 2.3)

%01000001

’a’

RecordPtr^.Field1[45,23]

@Var1

Abs(Var2)

Factors can have a unary operator (an operator that applies to only one value, unlike binary operators that apply to two). They

are:
@ This operator is the address-of operator. The returns
the address of the variable reference immediately
following. It is equivalent to the Adr() function.
- Sign negation. Takes the negation of the factor
following it.
+ Sign identity. It is accepted, but has no affect.
not Boolean or bitwise negation. If the factor is a
Boolean value, this returns the opposite. If it is an
integer type, it returns the bitwise compliment.
These factors are built into terms. Terms are either the same as
factors, or two factors with a multiplicative operator between
them. The multiplicative operations are the following:
* Multiplication
/ Floating point division. Both arguments to this
operator are converted to floating point before the
operation takes place.
div Integer division. Both arguments to this operator
are converted to integer forms before the operation
takes place.
mod The remainder operation. Both arguments are
converted to integers if necessary.
and Logical AND function. If both arguments are Boolean
values and the left factor evaluates to FALSE, this
operation will "short-circuit". Any Real factors are
converted to Integers.
shl Shift left. Shifts the bits in the left operand the
number of positions given by the right operand. Both
arguments are converted to Integers if necessary.
shr Shift right. Shifts the bits in the right operand
the number of positions given by the right operand.
This is a logical shift, not arithmetic, so the
left-most bit is assigned 0. Thus if you shift a
negative number in order to do division, you will get
nonsense results (use div instead - it will use shifts
when possible). Both arguments are converted to
Integers.
Terms, in turn, are built into simple expressions. Simple
expressions are either the same thing as terms, or two terms

grouped by an addition operator. The addition operators are:
+ Adds the two operands.
- Subtracts the second operand from the first.
or Logical OR. If both operands are Boolean values and
the left operand evaluates to TRUE, the right operand
is not evaluated. Both arguments are converted to
Integers if necessary.
xor Exclusive OR. Generates the left term exclusive ORed
with the right term. This never short circuits. If
either argument is a Real value, it is converted to an
Integer.
These simple expressions are built into complete expressions.
Expressions are either the same thing as simple expressions, or
two simple expressions grouped by a relative operator. These
operators produce Boolean results, and are defined as:
= Returns TRUE iff the two arguments are equal.
<> Returns TRUE iff the two arguments are not equal.
> Returns TRUE iff the left argument is greater than the
right argument.
< Returns TRUE iff the left argument is less than the
right argument.
>= Returns TRUE iff the left argument is either greater than
or equal to the right argument (that's only takes one
comparison, by the way).
<= Returns TRUE iff the left argument is either less than
or equal to the right argument.
(iff is shorthand for "if and only if")
Evaluation Order & Short Circuits

The normal rules of math are built-in to the grammar for
expressions. The unary operators are evaluated first, then all
the multiplicative operations, the additive operations, and
finally the relative operations. Thus the precedence looks like
this:
Operator Precedence

@,not,unary +/- First (highest)
*,/,div,mod,
and,shr,shl Second

+,-,or,xor Third

=,<>,<,>,>=,<= Fourth (lowest)

All expressions in parentheses are, of course, evaluated first.

Also, keep in mind that all of these operators are left associative - in other words, a factor between two operators of equal precedence is bound to the one on the left (e.g. "x+y+z" is treated as "(x+y)+z").

You should not make any conclusions about the order in which the two operands to a binary operator are evaluated. If you have the expression "function1 + function2" in your program, for example, there is no telling which function would be evaluated first.

The exception to that rule is for Boolean equations, but only if you are using short-circuit evaluations (the default). What is a short circuit? Imagine you have the expression "A or B" in your program. If the program evaluates A and finds that it is True, it already knows the final value of the entire expression. Therefore there's no point in evaluating B. The analogous case for "A and B" occurs when A evaluates to False. You can be sure, in a Boolean expression, that the operands of "and" and "or" will be evaluated left-to-right.

Short-circuit evaluations are often faster than normal ones, but they have the added benefit of protecting the programmer. For example, the statement:

if OpenMyWindow(w) and (w^.Width = whatever) then ....

... would be a problem without short-circuit evaluations because, for example, the variable w might not be defined if the function call fails. The value of the function would not change, but if w pointed to an odd address it could cause a guru. With short-circuit evaluations, the second factor would not be evaluated unless the first returned True.

Constant Expressions

Constant expressions are the same as normal expressions, but must be completely evaluated at compile time. Thus they can't use any external functions or variable references, but can use all the same operators and the standard functions (i.e. the ones built-in to the compiler).

Variable References

Variable references can get awfully complex, what with pointers
and records and arrays and so forth. The basic syntax is as
follows:
<Variable Reference> ::= <Variable Mark> |
<Variable Mark> <Selectors>
<Variable Mark> ::= <Variable ID> |
<Type ID> (<Variable Reference>)
<Selectors> ::= <Selector> | <Selector> <Selectors>
<Selector> ::= ^ | .<Field Id> | [<Index Expressions>]
Generally speaking, a variable reference will consist of the
identifier of a global or local variable, followed by any number
of selectors in any combination. The other possibility, which is
somewhat different as of version 1.2 of the compiler, is that you
can place a type cast somewhere in the middle of all this. The
following are all examples of variable references:
Var1
Var1^
Var1^.Field1[34,56]
Type2(Var1)^.Field2^^[23]
If you start writing code that looks like the last example, you
should get some rest.

1.15 Statements

==========
Statements
==========
Statements are the meat and potatoes of programs - they are what
goes between begin and end. They have the following form:
<Statement> ::= <If Statement> |
<While Statement> |
<Repeat Statement> |
<For Statement> |
<Case Statement> |
<With Statement> |
<Compound Statement> |
<Assignment Statement> |
<Procedure Call> |
<Return Statement> |

<Goto Statement>

If Statements

The if statement allows you to test alternatives. It has the following form:

<If Statement> ::= if <Boolean Expr> then <Statement> |
if <Boolean Expr> then <Statement>
else <Statement>

If the Boolean expression evaluates to True, the statement following the word "then" is executed. If the Boolean expression evaluates to False, the statement in the else clause, if one exists, is evaluated. If no else clause exists, execution continues with the statement following the "if" statement.

There is an ambiguity here - consider the following fragment:

if Expr then if Expr then Statement1 else Statement2

To which "if" test does the else clause apply? PCQ, like most implementations, always attaches the else part to the most recent "if".

While Statements

The "while" statement is one of several that allow you to loop. It has the following form:

<While Statement> ::= while <Boolean Expr> do <Statement>

At the beginning of the loop, the Boolean expression is evaluated. If it evaluates to True, the statement part is executed. When that finishes, the expression is re-evaluated, and the process continues until the expression evaluates to False. Once that happens, execution continues with the statement following the while statement.

Repeat Statements

The "repeat" statement provides a slightly different form of looping. It has the following form:

<Repeat Statement> ::= repeat <Statements> until <Boolean Expr>
<Statements> ::= <Statement> | <Statement> ; <Statements>

When the repeat statement is encountered, the statement part is executed, then the Boolean expression is evaluated. If the expression evaluates to False, the statement part is executed again. If the expression evaluates to True, execution continues

at the following statement. Note that "repeat" loops always execute at least once, whereas "while" loops might not execute at all. Also note that the <Statements> part can actually be empty.

For Statements

The "for" statement provides a third type of looping, for occasions when you know exactly how many iterations of the loop you want to execute. It has the following form:

<For Statement> ::= for <Variable Reference> :=
<Expression> <Direction>
<Expression> do <Statement>
<Direction> ::= to | downto

Note that this is different from the PCQ version 1.1. When a "for" loop is encountered, the first expression (the initial value) is evaluated, and its value is stored in the variable reference (called the index). Then the index is compared to the second expression (the final value). If the direction is "to", and the index is less than or equal to the final value, the statement part is executed. At the end of that, the index is incremented by one and again tested against the final value. This goes on until the index is greater than the final value, at which time execution moves on to the next statement.

If the direction is "downto", the same thing happens, except the index is decremented at each pass, and looping continues until the index is less than the final value.

Note that the "for" loop might not execute at all, if the initial value is larger than the final value (for "to" loops, or vice versa for "downto" loops). Also note that the final value is fully evaluated each time through the loop, so if it’s a complex expression you might want to store it in a variable before the "for" statement.

You can always use a "while" or "repeat" loop in place of a "for" loop, and in fact the "for" loop is least often used. Niklaus Wirth, the guy who designed Pascal and Modula-2, actually left the statement out of his latest language (Oberon).

It is considered an error to modify the index variable within the loop, but PCQ Pascal doesn’t enforce it. That doesn’t make it a good idea, of course.

Case Statements

The Case statement is used to test a value against a series of alternatives, and process instructions accordingly. It has the following form:

<Case Statement> ::= case <Expression> of <Alternatives> end |
case <Expression> of <Alternatives>
else <Statement> end

<Alternatives> ::= <Alternative> | <Alternative> ; <Alternatives>

<Alternative> ::= <Case Vals> : <Statement>

<Case Vals> ::= <Case Val> | <Case Val> , <Case Vals>

<Case Val> ::= <Const Expr> | <Const Expr> .. <Const Expr>

When the case statement is executed, the expression is evaluated. It is then compared against all of the cases and ranges until it matches one, at which point the associated statement is executed. If none of the alternatives match, the else statement is executed, if there is one. If not, execution just continues with the statement following the case statement.

Note that the expression is only evaluated once, so a case statement can be more efficient than an equivalent series of "if" statements. Also note that one statement (at most) from the case statement is executed - it’s not like C, where execution falls through until you tell it to knock it off.

With Statements

The With statement allows to specify a record variable to which a statement will apply. Within the statement you can abbreviate references to the record’s fields by specifying just the field name. That saves you some typing, and the compiler generates somewhat more efficient code. It serves no functional purpose.

The format of a With statement is as follows:

<With Statement> ::= with <Expressions> do <Statement>

<Expressions> ::= <Expression> | <Expression> , <Expressions>

If you specify more than one record (separated by commas), the compiler treats it like nested with statements, with the first record listed at the outermost scope and the last one at the innermost.

In the statement, any references to fields will be considered to apply to the innermost record to which it applies. Ambiguities can be cleared up by using the full record specification.

With statements are especially handy when the record reference is long and convoluted - it will only be executed once, which increases efficiency. The other time it comes in especially handy is filling in values in a record. It doesn't always improve the program's efficiency, but it saves a lot of typing.

If you are executing under a with statement of the form "with pˆ do ...", it is considered an error to alter the value of p. For example, it is an error to free the memory of a record in a "with" statement referencing the record. This error is not detected by PCQ Pascal, but it's a good idea to avoid it.

Compound Statements

The compound statement allow you to use a series of statements in any situation that calls for at least one statement. It has the following form:

<Compound Statement> ::= begin <Statements> end

A compound statement can in fact be used wherever a statement is allowed, and the compound statement by itself generates no extra code. In other words, surrounding a group of expressions with "begin" and "end" will not change the code the compiler produces - it just clarifies your purpose.

Assignment Statements

Assignment statements are used to set the value of variables. They have the following form:

<Assignment Statement> ::= <Variable Reference> := <Expression>

Remember that the type of the variable must match the type of the expression identically (see the section called Type Declarations for the specific rules).

Procedure Calls

Procedure calls transfer execution to a pre-defined routine, passing parameters as required. They have the following form:

<Procedure Call> ::= <Identifier> |
<Identifier> (<Actual Params>)

<Actual Params> ::= <Actual Param> |
<Actual Param> ; <Actual Params>

<Actual Param> ::= <Expression>

Procedures are different from functions in that functions return a

value, and are callable only from expressions. Procedures do not return a value, and are callable only from statements.

When a procedure statement is executed, each actual parameter is evaluated and pushed on the stack, in left to right order.

Because C uses variable numbers of parameters, it pushes actual parameters on the stack in right to left order. Therefore if you plan to call a C routine you will need to reverse the order of the parameters.

Return Statements

Normally, execution of a procedure continues until the end of the procedure, when execution returns to the caller. The "return" statement allows you to escape the current procedure immediately. It has the following very simple form:

<Return Statement> ::= return

The "return" statement is only valid within procedures. It is not valid within functions because they must return a value - the analogous statement in a function is an assignment to the function identifier, which causes the function to return immediately. The "return" statement is also not valid in the main body of the program, because it would cause the program to terminate. Instead you must explicitly terminate the program with the Exit statement.

Goto Statements

The "goto" statement immediately transfers execution to a defined label. It should be used with extreme care, and if possible avoided. Its form is:

<Goto Statement> ::= goto <Identifier>

Standard Pascal requires that a goto label should be a series of digits, and Turbo Pascal extends that definition to allow identifiers to be used as labels. PCQ Pascal takes the next logical step by requiring that labels be identifiers, not numbers.

Most programmers will tell you that the "goto" statement is simply bad programming practice, because it makes programs difficult to understand and debug. There are, however, other problems. For instance, if you "goto" from inside a "for" loop to outside of the loop, you will confuse your program’s stack (until you leave the routine). Similar things happen for the "with" statement. I won’t even bother to tell you the havoc that would ensue if you jumped into a "for" loop from outside.

The moral of the story is: try to avoid them.

1.16 Input / Output

==========

Input/Output

==========

Pascal IO

You can think of the Pascal IO routines as an interface to the
AmigaDOS routines. The Pascal IO routines all use AmigaDOS for
the actual reading and writing, so if you felt like it you could
just use an AmigaDOS file instead.
Pascal IO does have several advantages, however. First of all,
all PCQ Pascal files are buffered, which means that reads and
writes do not necessarily have to access AmigaDOS each time. That
speeds things up a bit.
The other advantage you get is the use of the Pascal Read, ReadLn,
Write and Writeln routines, which can be awfully handy.
When you declare a file in your program, as either a "File of Some
Type" or as "Text", you are in effect allocating a record. That
record, if its fields were accessible, would look like:
PCQFile = record
Handle : An AmigaDOS file handle
Next : A pointer to the next open Pascal file
Buffer : The address of the file's buffer
Current : The position in the buffer at which the
next read or write will take place.
Last : The Last position of a read
Max : The address of the end of the buffer+1
RecSize : The size of the file elements
Interactive : A Boolean value - True means the file
is attached to a CLI window.
EOF : A Boolean value
Access : A Short value, either MODE_NEWFILE or
MODE_OLDFILE
end;
Text and Typed Files

Pascal has two very different types of files. The normal kind of
file is called a typed file, and is declared with a type

specification of the form:
FileVar : File of ElementType
Typed files are essentially unlimited sequences of elements of that one element type. The Write command can be used to write individual elements to the end of the file, and the Read command can be used to input the current element and move the file pointer to the next one. You can only write one type of element to each typed file.
Typed files store their elements in exactly the same format that they appear in memory, so they are often unreadable. A file of character elements is readable, but a file of integer elements would be incomprehensible. Since typed files always store complete elements, each element is necessarily a fixed size.
You don’t actually need anything but typed files - anything you can do with Pascal files can be done with them. For the sake of convenience, however, Pascal also uses the Text file type. The Text file is a special kind of "File of Char", but is not compatible with it. Text files are made up of ASCII characters broken up into variable length lines. Each line is terminated by a special newline character, which on the Amiga is the line feed character (ASCII 10. On MS-DOS machines, the newline character is the carriage return/line feed sequence).
Since Text files (like this documentation, and all the source files) are so common, Pascal has a variety of routines built-in to read and write values to and from them. Most of the standard types, for example, have a special routine for converting them into ASCII characters and writing them to a file. These routines are called automatically by the compiler to write each expression in a Write procedure to a Text file. Similar routines exist for reading the same types. The descriptions of the Read and Write procedures below described exactly how this conversion takes place.
There is even a special form of the Read and Write procedures designed to work only with Text files - the ReadLn and WriteLn commands. They are explained below, but note that since typed files have no lines, these two routines are used only for Text files.
Opening Files

There are several closely related routines for opening Pascal files. For opening input files, which are existing files that will be opened for reading, you can use either the ReOpen() function or the Reset() procedure. They are defined as follows:

Reset(var FileVar : File or Text;
FileName : String;
BufferSize : Integer);

The "Reset" procedure opens a file for reading. The FileVar parameter specifies the Pascal file variable that should be associated with this AmigaDOS file. The FileName is the file name, with a complete path if necessary. Since this can be any valid AmigaDOS filename, it can actually refer to the printer, a console window, or all sorts of other things.

The BufferSize parameter specifies the size of the buffer you want, in bytes. When PCQ tries to open your file, it will try to allocate this much memory as a buffer. If it can't, it sets IOResult and fails. Actually, PCQ tries to allocate a buffer that is an even multiple of the file element size, not larger than BufferSize, but at least one element long (the element size for Text files is one character). The BufferSize parameter is entirely optional - if it's not included, a default value of 128 is used.

If you are using automatic IO checking, your program will abort with a runtime error if the file cannot be opened. If you are checking manually, it will just set IOResult.

The variable FileVar should not refer to an file that's already open. Either the file should be closed first, or you should use a different file variable.

ReOpen(fname : String;
var fvar : file or text;
buffersize : Integer) : Boolean

ReOpen is the function form of Reset. It attempts to open a file in the same way as Reset, and if everything goes O.K. it returns True. If there was a problem, it returns False.

ReOpen never sets IOResult, so if you are using automatic IO checking a call to ReOpen will not abort the program. As in the Reset procedure, the BufferSize parameter is optional.

If you need to open a file for writing (which deletes any existing file of the same name!), you can use the Rewrite procedure or the

Open() function. They are defined as:
Rewrite(var FileVar : File or Text;
FileName : String;
BufferSize : Integer);
Rewrite opens an AmigaDOS file for writing, erasing any existing file with the same name. The parameters are just like the Reset command, and the BufferSize is still optional.
Rewrite sets IOResult, so if your program is using automatic IO checking it will abort if the file cannot be opened. Programs handling their own IO checks should be sure to look at IOResult.
As with all the file opening routines, you should not use a file variable that already refers to an open file.
Open(fname : String;
var fvar : file or text;
buffersize : Integer) : Boolean
Open is equivalent to Rewrite, but in function form. It attempts to open the file for writing, and if everything goes O.K. it returns True. If there is any problem, it returns false.
Remember that opening a file for output means that any existing file with the same name gets erased.
Open does not set IOResult, so programs using automatic IO checking can use it without aborting. The BufferSize parameter, just as in the rest of the file opening routines, is optional.
Writing to Files

Output in Pascal is handled by the Write and Writeln procedures. Unlike most Pascal procedures, these routines will accept an unlimited number of parameters. If you are writing to a Text file, the parameters can be a variety of types. The Write procedure has the following basic form:
Write(var FileVar : Text or Typed File, Expressions...)
The FileVar is optional. If it is not included, the standard Text file Output is used in its place. If the FileVar is a typed file, each expression must be of the element type. They will be written to the file with no space in between them, formatted just as they are in memory.
If you are writing to a text file, it’s a different story entirely. First of all, each expression can take the form "e:m:n" where e is the expression itself and m and n are constant

integers. m specifies the minimum field width - if the item to be written would take up fewer than m characters, spaces are written to fill out the field. n is only allowed if e is an expression of type Real, and it specifies how many digits to the right of the decimal point will be written out. The default values for m and n are 1 and 2, respectively. Specifically what gets written out depends on the type, according to the following:

Integer

Short

Byte These three types are all written as sequences of ASCII digits, as expected. They are not preceded by any spaces, but Short and Integer types might lead off with a minus sign if appropriate. There will be no trailing spaces.

Real The entire integer part (the part to the left of the decimal point) is written out. If n is not zero, a decimal point followed by n fractional digits are written out. Note that for Real numbers, m indicates the number of digits to the left of the decimal point that should be written, not the full field width (this will probably change in the next version).

Char A character is written out as just the single character, unchanged.

Array of Char

The entire array, for its whole declared length, is written out character by character.

String The entire string up to, but not including, the terminating null character is written.

Boolean Either the word TRUE or the word FALSE is written, with no leading or trailing spaces.

WriteLn(var FileVar : Text; Expressions...)

Writeln is exactly the same as Write, but after is has written out each of the expressions according to the rules above, it also writes out a newline character. WriteLn can only be used on Text files.

Reading from Files

Input in Pascal is handled by the Read procedure. Like the Write procedures, Read accepts any number of parameters, and many types of parameters if you are reading from a Text file. The format is

as follows:
Read(var FVar : Text or Typed File; Variable References....)
The FVar parameter is optional. If it is omitted, the standard
Text file "Input" is used in its place.
If FVar refers to a typed file, each variable reference is filled
with the subsequent file elements, unless EOF is reached or some
other error occurs first. Reading from a Text file is a somewhat
more flexible job. Just as with the Write statements, what
happens depends on the type being read, according to the
following:
Integer
Short
Byte All white space, which is defined as any character whose
ASCII value is less than or equal to 32 (the space
character) is skipped. If the first non-white space
character is not a digit or the EOF is reached, an IO
error is raised (see Exit Procedures for more
information). If not, digits are read until the first
non-digit. The resulting number is returned, and the
file pointer rests on that non-digit.
Real First, an integer is read exactly as above. If the next
character is a period, digits are again read in as
fractional digits until the first non-digit. The pointer
rests on that first non-digit. A Real value does not
need a fractional part, nor does it require a decimal
point.
Char Reads the next single character. Standard Pascal
translates end-of-line characters into spaces, but PCQ
Pascal, like Turbo Pascal, allows them to come through as
ASCII linefeeds.
Array of Char
Reads characters into the array until either the array is
full, or the end-of-line is reached. If the end-of-line
is reached, the rest of the array is padded with spaces
and the file pointer is left pointing at the end-of-line
character - you'll need to call Readln to get rid of it.
String Reads characters into the String buffer until the
end-of-line is reached. At that point the string is
terminated by a zero byte, and the file pointer is left

pointing to the end-of-line character. Note that this
routine does not check for length, so you need to be sure
to provide a large enough string for the file you are
reading.
Boolean Can't be done.
A series of variable references in a Read procedure acts exactly
like each of the references in individual Read procedures. Thus:
Read(Input, Var1, Var2);
is equivalent to:
Read(Input, Var1); Read(Input, Var2);
which is also equivalent to:
Read(Var1); Read(Var2);
ReadLn(var FVar : Text; Variable References...)
Readln is exactly like Read, but after all the specified variables
have been read, this command eats characters until it finds the
next end-of-line, and it eats that too. This command can only be
used on Text files.
File Buffers

Like Standard Pascal, but unlike Turbo Pascal, PCQ allows you to
access the file buffer directly. The syntax of the reference
looks like:
<File Variable Reference> ^
e.g. FileVar^
That expression has the same type as the elements of the file
itself (Text files have Char elements). With input files, this
syntax looks at the next element that will be input by the Read()
procedure, like a look-ahead. If EOF(FileVar) is True, then
FileVar^ is invalid.
For output files, the file buffer reference shows you the output
buffer. This is not the value that will be written out with the
Write() procedure - see the description of Put() below for more
information.
Logically, the file buffer you access through this syntax is only
as long as one element, but the actual buffer in memory can be of
virtually any size. See the file opening commands for information
on setting the buffer size.
Using this syntax in PCQ is actually considered an IO operation,
in the sense that it can set IOResult. This only occurs when the

file is interactive, because the buffers of interactive files are
not kept full.
If you are making use of the file buffer, you will probably use on
or both of the following routines:
Get(var FileVar : Text or Typed File)
Get just moves the file pointer from the current element to the
next one. That may involve calling DOS to refill the buffer, but
normally it's just a matter of adjusting a field in the file
variable itself. It is an error to call this routine on a file
for which EOF(FileVar) is True.
Every read from a typed file of the form "Read(FVar,Element)" can
be thought of as:
Element := FVarˆ; { To assign the current value }
Get(FVar); { To advance the file pointer }
Put(var FileVar : Text or Typed File)
Put moves the file pointer past the current element in an output
file, writing the buffer to disk if necessary. Every statement of
the form "Write(FVar,Expr)", where FVar is a typed file, could be
implemented as:
FVarˆ := Expr; { Set the buffer value }
Put(FVar); { Commit the buffer }
Note that until the Put procedure is called, any assignments to
the file buffer will write over each other.
IO Checking

Input & Output can cause more unforeseen errors than virtually any
other area of programming. PCQ Pascal catches IO errors in one of
two ways. The first is called automatic IO checking - PCQ checks
for errors after each and every IO operation. If there is an
error, the program aborts with a runtime error. Believe it or
not, that's the default behavior.
If you turn off automatic checking using the {$I-} compiler
directive, you become responsible for checking for errors. You do
this by calling the IOResult function, which returns 0 if
everything is O.K., or some other number if there was a problem.
When you call IOResult, you automatically clear its value, so if
you are going to use the error code later on you need to save it
in a variable.
IOResult is set by every IO routine. No IO routines will function

if IOResult is not zero, so you should check for errors whenever possible to avoid skipping reads and writes.

The values returned by IOResult are the same as the runtime errors - possible values are listed in the section called Exit Procedures.

Standard IO

One of the tricky parts of programming on the Amiga is that a program can be launched from two very different environments. The CLI invokes a program in much the same way as MS-DOS does - the program has obvious input and output channels. The Workbench, on the other hand, offers nothing of the sort. A program that needs input and output channels must establish them explicitly.

All PCQ Programs establish some sort of standard input and output channels, which are accessed through the Text files Input and Output. What those files represent depends, naturally, on the environment from which the program was executed.

Let's take the simple case first. If you run a program from the CLI, Input and Output will refer to the CLI window itself. Unless you use indirection, that is, in which case Input and Output will refer to the channels specified on the command line (see an AmigaDOS manual for more information). Maybe that wasn't so simple after all.

If the program was launched from the Workbench, the startup code goes through several steps to establish Input and Output. First of all, it looks at the value of the standard String typed constant StdInName. If StdInName is Nil, no file is opened and referring to Input will cause grave problems. If not, the startup code attempts to open the file. If it can't do it, it terminates with run-time error 53. If all went well, Input is established.

Next the startup code looks at StdOutName. If it's Nil, Output is not established, so any Write procedures had better use an explicit file. If StdOutName refers to the same string as StdInName (not two strings with the same contents - it actually has to point to the same memory), and if the file associated with Input is interactive (i.e. attached to a console window), then Output is associated with the same file handle as Input. The default values of StdInName and StdOutName have this property. If either of those tests fail, the startup code tries to open a file

according to StdOutName, and if it works, then Output is set up
accordingly. Otherwise, the program will terminate with runtime
error 57.
The default values for the two channels are defined in PCQ.lib as
follows:

```
CONST
StdInName : String = "CON:0/0/640/200/";
StdOutName : String = StdInName;
```

According to the rules specified above, this would normally result
in one full-screen console window being opened, with input coming
from it and output going to it.
Note that StdInName and StdOutName are defined as typed constants.
They can't be variables, because they are opened before any of
your code executes. If that's a major problem, you can define
them both as Nil, and from inside your program you can open Input
and Output as anything you like. You should probably re-use Input
and Output only when your program was run from the Workbench (you
can check if you were run from the Workbench by calling
GetStartupMsg from "Utils/Parameters.i").
If you compiled your program using the "-s" small initialization
code switch, all bets are off: Input and Output are not
established, and in fact none of the other IO routines are
initialized either. See the section called Small Initialization
Code for more information.

1.17 Strings

```
=====================================================================
Strings
=====================================================================
```

PCQ Pascal strings are similar to C strings, and nothing at all
like Turbo Pascal strings. That's too bad, because Turbo Pascal
strings are a lot easier to work with. Eventually, PCQ Pascal
strings will be the same as Turbo Pascal strings, but that day has
yet to come.
The string type can be thought of as simply "^Char". Thus all
string variables take up exactly 4 bytes, and they store the
address of the actual text. The text can be of any length, and
is terminated by the character Chr(0). A statement like:

StringVar := "A String"
... actually stores the address of the constant string in the
string variable, rather than copying the string itself into some
existing space.
String constants in the Pascal text are delimited by double quote
characters, as opposed to the single quote characters of character
array constants. Thus "A String" is considered a constant of type
String, whereas 'not a string' is considered a constant of type
"Array [0..11] of Char".
Strings are defined as simple pointers, so the reference
"StringVarˆ" is valid, and is of type Char. It actually points to
the first byte within the string buffer. But unlike normal
pointers, the individual characters in a string can also be
accessed through subscript notation. For example, to get the
fourth character in a string, you could use "StringVar[3]".
Remember that "StringVar[0]" is the first element.
Allocating String Space

Since the actual String variables only store an address, you need
to allocate some space for the actual text. C allows you define
that space when you declare the string type, but PCQ Pascal
doesn't even offer that much. In PCQ Pascal, you have to
explicitly allocate the memory from the system, or assign the
address of a buffer variable to the string. There are several
functions, all defined in the include file StringLib.i, that
allocate space for strings. They are:
AllocString(Size : Integer) : String
AllocString simply returns an uninitialized memory area that is at
least Size bytes long. This memory is considered PCQ memory, so
it is returned to the system after its use (see the section called
Memory Management for more information).
strdup(InString : String) : String
The strdup (string duplicate) function is equivalent to the
following:
temp := AllocString(strlen(InString));
strcpy(temp,InString);
strdup := temp;
In other words, it allocates just enough memory to hold the input
string, then makes a copy of the input string in the new buffer.

This memory is also considered PCQ memory, so it will be returned
to the system when the program terminates.
The other way to set up a string buffer is to assign the string
variable the address of an array declared elsewhere. Thus the
following:

```
Var
String1 : String;
Buffer1 : Array [0..127] of Char;
begin
String1 := Adr(Buffer1);
```

... sets up a 128-character buffer for String1. If you try to
store a 129-character string in that buffer, all sorts of problems
will result, so take care.
Using Escape Characters

There are, of course, lots of useful characters that can't easily
be written into a program source file. The end-of-line character,
for example, is always considered the end of a line, rather than
taken as a constant. If you need to include this sort of
character in any text constant (a Char, Array of Char, or String),
you use the C escape convention, in which a backslash "\" is
followed by a special character. The escape sequences supported
by PCQ are:
\n Line Feed, chr(10)
\t Tab, chr(9)
\0 Null, chr(0)
\b Backspace, chr(8)
\e ESC, chr(27)
\c CSI (Control Sequence Introducer), chr($9B)
\a Attention, chr(7)
\f Form Feed, chr(12)
\r Carriage Return, chr(13)
\v Vertical Tab, chr(11)
Every other character passes through unchanged, so the following
definitions also hold:
\ Just a single backslash
\' A single quote, even in a single-quote-delimited
constant
\" A double quote, even in a String constant

Thus a string like "A\tboy\nand\his \"dog." becomes:
|A boy
|and\his "dog
... where the vertical bar | represents the left margin.
StringLib

StringLib.i is an include file in the Utils directory that defines
a bunch of C-style functions for use on strings. Those functions
are explained in the include file itself. The source for these
routines is in the runtime library source, which is available in
the registered version.

1.18 Memory management

==========
Memory Management
==========
One of the problems with Amiga programming is that there is no
resource tracking. In an MS-DOS program, you can open files and
allocate memory like crazy, then just quit the program and
everything's O.K. On the Amiga, all those files would stay open
and the memory would stay allocated. You are expected to clean up
after yourself.
To help you do that, PCQ Pascal uses a special memory allocation
scheme that uses the Intuition AllocRemember routine to keep track
of all your memory allocations. When your program terminates, any
memory you haven't deallocated is automatically freed by an exit
procedure.
This only works for memory allocated through one of the special
PCQ routines New, AllocString (from Utils/StringLib.i) or GetMem
(from Utils/PCQMemory.i). Memory allocated through one of these
routines is called PCQ memory, but of course it's no different
from normal system memory except that it will be returned.
Even though it will be automatically freed, you should deallocate
memory as soon as you are through with it. There might be other
programs running that can use it. If you do free PCQ memory, you
need to use Dispose, FreeString or FreePCQMem to do it. If you
don't, the memory will be returned to the system, but the PCQ
memory routines won't know about it. They will, therefore, try to

free it again when the program terminates. Guru.
PCQ memory is always allocated with the flags MEMF_PUBLIC and
MEMF_CLEAR. If you need chip memory, therefore, you should use
the normal Exec function AllocMem (from Exec/Memory.i).
HeapError

So what happens when the PCQ memory routines are unable to
allocate some memory a user has called for? That depends on the
value of HeapError, a standard Address variable. HeapError
contains the address of a function that has a header with the
following form:
Function HeapFunc(Size : Integer) : Integer;
The Size parameter is the amount of memory the allocator was
trying to get, and the integer returned determines how the
allocator will respond to the problem. If it returns 0, the
allocator will abort the program with runtime error 54. If it
returns 1, the allocator will return the value Nil, so you'll have
to check after each call to New, AllocString or GetMem for a Nil
value. If the HeapError function returns 2, the allocator tries
the allocation again. This gives you a chance to free up some
memory, if possible. If that allocation fails, the HeapError
function will get called again.
To install a function as the HeapError function, you would use
something like the following:
HeapError := @HeapFunc;
... where HeapFunc is defined as above.

1.19 Exit procedures

==========
Exit Procedures
==========
Exit procedures are routines that you set up to run after the main
routine is finished, or if there is a run-time error. They are
used to return resources to the system, or to exit gracefully or
even recover from run-time errors.
How it works is: when a program terminates, for whatever reason,
PCQ examines the value of ExitProc, a standard Address variable.
If it is non-nil, PCQ sets ExitProc to Nil and calls the procedure

it pointed to. When the procedure returns, PCQ checks ExitProc again, and continues calling exit procedures until ExitProc is Nil. The normal PCQ initialization code sets up an exit procedure that frees all memory allocated through New or AllocString, and closes any Pascal files that remain open. Normal programs, therefore, have at least one exit procedure.

To make a procedure an exit procedure, first define a routine with no parameters. Then you set the ExitProc standard variable to the address of your routine. In most cases you'll want to save the previous value of ExitProc so all the other exit procedures can run as well. Within the exit procedure itself, you should reset ExitProc to point to the previous routine.

There are two additional variables associated with exit procedures that are only valid while the procedure is actually running as an exit procedure. They always exist, but their value is only set when the program terminates. They are:

ExitCode ExitCode contains the value of any runtime error that caused the program termination. This is either a value you supplied through the Exit() routine, or a runtime error code.

ExitAddr ExitAddr is the address where a runtime error occurred. It is only valid when ExitCode is non-zero.

It can (in theory) be used to recover from a runtime error, but you'd have to be awfully familiar with the actual runtime code to get it to work. The runtime source code comes with the registered version of PCQ Pascal.

Runtime Errors

There are several errors that will cause a PCQ program to immediately terminate. When that happens, ExitCode will be set to a specific value, which will eventually be returned to AmigaDOS. AmigaDOS, in turn, will normally ignore it, so you'll never see it. The two ways I know of the see the return value (if it is non-zero) are to run the program in a script, or to run the program using AmigaDOS's Run command. In either case, if the program has a non-zero return code AmigaDOS will write out "ProgramName failed returncode #".

The runtime errors generated by PCQ programs are as follows:

Code Description

---

50 No memory for IO buffer (1)

51 Read past EOF (1)

52 Input file not open (2)

53 Could not open StdInName (3)

54 New() failed (4)

55 Integer divide by zero

56 Output file not open (2)

57 Could not open StdOutName (3)

58 Found EOF before first digit while

reading an integer (1)

59 No digits found in reading an integer (1)

60 Range error (5)

The following AmigaDOS error codes can result from a call

to Open, ReOpen, Write, Read, etc: (1)

103 Insufficient free store

202 Object in use

203 Object already exists

204 Directory not found

205 Object not found

206 Invalid window specification

210 Invalid component name

212 Object wrong type

213 Disk not validated

214 Disk write protected

218 Device not mounted

221 Disk full

223 File is write protected

224 File is read protected

225 Not a DOS disk

226 No disk in drive

(1) These errors only cause runtime errors when automatic

IO error checking is enabled (it's the default). If

you have used the {$I-} directive to indicate that

you'll check IOResult explicitly, the program will

not automatically terminate.

(2) PCQ cannot always determine when the file is not

open. If it can, it issues this error. If it can't,

the machine will crash.
(3) These errors occur in the initialization code, before
any exit procedures are established.
(4) See the section called Memory Management to see how
keep a runtime error from being issued.
(5) This only occurs if you have turned range checking on
with the {$R+} directive.

1.20 Compiler directives

==========
Compiler Directives
==========
There are several options that PCQ Pascal provides that are not
useful for all programs. For example, a large application should
not terminate with a runtime error if it can't open a file.
Therefore PCQ Pascal allows you to determine several aspects of
the program generated from your code.
You control these options by issuing compiler directives.
Compiler directives are contained in comments, and must begin with
a dollar sign "$" as the first character in the comment.
Immediately following the dollar sign is a letter indicating the
directive, which is then followed by zero or more characters
giving additional information.
You can include more than one directive in a single comment by
separating the directives by commas. Thus the following
directives turn IO checking off, and range checking on:
{$I-,R+}
The following directives are supported:
{$A Any number of assembly instructions}
The $A directive allows you to insert assembly language
instructions into the source code at the given point. The text of
the instructions is passed through to the assembler unchanged, so
you should include comments and make variable references according
to assembly language style.
Since comments are allowed anywhere that white space is allowed,
you could technically insert some assembly language instructions
in the middle of an expression. That could produce unpredictable
results, however, so you should use assembly language only between

statements.

{$B+} or {$B-}

The $B directive turns short-circuit evaluations on and off. If you use the $B+ directive to turn short-circuit evaluations on, which is actually the default, PCQ will evaluate Boolean expressions normally, but as soon as the final value of the expression is known, it will skip any remaining parts of the expression. Thus in a series of "and" clauses, as soon as one of them evaluates to False, the rest are not evaluated at all. In a series of "or" clauses, as soon as one of them evaluates to True, the rest are skipped. If you are using short-circuit evaluations the expression will always be evaluated from left to right. Short-circuits make Boolean evaluation somewhat faster, especially in long equations.

If you turn short-circuit evaluations off using the $B- directive, all parts of a Boolean expression will be evaluated even if the outcome is not in doubt, and the expression will not necessarily be evaluated in left-to-right order.

{$I "fname"}

This directive inserts the file "fname" into the input stream at the current position in the file. When the included file has been fully read, the input is again taken from the original file. You can’t use any other directives after this one, but you can include any comments you like after the file name.

In order to keep from including a file more than once, the compiler keeps a list of all file name already included. If the file name matches one already on the list, it is not included. Note that only the actual file name, not the entire path, is compared. Thus you should be sure that all your include files have unique names.

Almost all the example programs demonstrate the use of include files.

{$I+} or {$I-}

The other form of the $I directive determines your programs reaction to Input/Output errors. The default behavior, which corresponds to the {$I+} directive, is to issue a runtime error whenever an error is detected in an IO routine. The other option, specified by the {$I-} directive, indicates that the program itself will check the IOResult function after IO operations, and

handle any errors accordingly. See the Input/Output section for more information.

{$SN} or {$SX} or {$SP} or {$SD}

The $S option controls the storage of global variables and typed constants. The N option (for Normal storage) tells the compiler that for all subsequent global variables, the compiler should allocate memory in the data segment, and also issue an external definition (XDEF in assembly language) for the identifier, so it can be used by external routines. This is the normal storage scheme for PCQ programs.

The X option (for eXternal storage) tells the compiler that all subsequent global variables are defined outside of the program. Therefore the compiler should not allocate any space for the variables, but simply create an external reference (XREF in assembly language). This is the normal storage scheme for external files.

The P option (for Private storage) tells the compiler that it should allocate space for global variables, but it should not export the identifier itself. This allows external files to have global variables that do not affect the main program.

The D option (for Default storage) resets the storage to its standard value. In an external program, it acts like {$SX}, and in a normal program it acts like {$SN}.

{$O+} or {$O-}

These options are identical to the {$I+} and {$I-} options.

{$R+} or {$R-}

The $R directive determines whether the compiler will verify that index values are within the specified range of the array. The default behavior is {$R-}, which means that the compiler will not generate the extra code. Specifying {$R+} makes the compiler issue a runtime error with error code 60. Turning this option on results in larger and slower programs, so I would recommend only using the option while testing.

1.21 Type casts

==========

Type Casts

==========

If you are using a strongly typed language like Pascal, you need a way to get around the type rules. PCQ Pascal uses the same method as Turbo Pascal and Modula-2 to get around the type checks. The format for a type cast is as follows:
<Type ID> (<Expression or Variable Reference>)
This looks just like a function, but it is definitely not one. A type cast never generates any code - it just lets the code get by the compiler. Naturally, this can lead to some serious problems. For example, a type cast like "Short(EnumeratedVar)" will cause problems because enumerated types are normally one byte long. You have told the compiler to consider it a two byte value, but the other byte is undefined. The correct way to handle that call would be to use the Ord() function, or at least cast it to a Byte instead.
The difference between Short() and Ord() in this example is that Ord() is a type transfer function, whereas Short() is not a function at all. As another example, consider the fragment below:
Writeln(Integer(2.0));
Writeln(Trunc(2.0));
Those two lines will not write the same values. Trunc(), another type transfer function, converts the real value 2.0 into the integer value 2 before writing it. The Integer "function" does nothing at all, so the program just writes the real value as if it were an integer, which is sure to produce some preposterous number.
In general, casting an expression to a larger sized expression is a bad idea, and casting a simple expression (an ordinal, real, or pointer value) to a complex type (an array or record reference) or vice versa will almost always produce nonsense results.

1.22 Small initialisation code

==========

Small Initialization Code

==========

You might, sometime, want to create a program that doesn't have the overhead of normal PCQ Pascal programs. For example, if you never use Pascal files, there's no point having all the code for Writeln() in your program. If you want to cut the start-up and

shut-down code that PCQ adds to your program to a minimum, you can use the "-s" option (for Small) on the command line. For example, to compile Tiny.p with the small startup code, you would use the following line:

Pascal Tiny.p Tiny.asm -s

The overhead you get rid of is not useless fat, however, and one of the problems with not having it is that you can't use it. Put another way, if you use the "-s" option on the command line, you will not be able to use any Pascal IO routines. Specifically, the procedures and functions that become off limits are:

Write, WriteLn, Read, ReadLn, Get,
Put, Open, ReOpen, Close, IOResult

The compiler will flag references to all these identifiers with the exception of IOResult, but keep in mind that the compiler will never see references in separately compiled files (external file) or object code libraries. The moral of this story is that you should exercise extreme care when using this option, or you will end up linking both startup routines and crashing the machine anyway.

So what, exactly, is missing? The normal startup code does the following:

1. Handles the CommandLine or Workbench message.
2. Opens Intuition, DOS, and the MathFFP library.
3. Sets up Input and Output, opening a window if necessary.
4. Initializes the memory allocation list and the open file list so they can be freed at exit.
5. Sets up an exit procedure that will free up the memory and files.

As you can see, there are several pretty big routines: Open(), Close(), setting up Input and Output based on the way the program was run and the contents of StdInName and StdOutName, the exit procedure that frees everything, etc. The small initialization code does this:

1. Handles the CommandLine or Workbench message.
2. Opens Intuition, DOS and the MathFFP library.
3. Intializes the memory list, and sets up a simple routine to free all the New() memory.

And that's it. According to my preliminary results, this program: "Program Test; begin end." compiles to about 2.5k normally, and less than 700 bytes using the small initialization code.

1.23 External files

==========

External Files

==========

In developing the compiler I found that writing the entire thing
in one source file was just too much - the intermediate assembly
files were huge, and the compile times were ridiculous. In order
to split things up a bit I added a method for separate
compilation, similar in purpose to Turbo Pascal's units, but less
powerful.

External files have the following format:

<External File> ::= External; <Definitions>

In other words, an external file starts with the reserved word
External, a semicolon, then any number of definition blocks.
These definition blocks, as defined above, can be functions,
procedures, variables, etc.

Whenever you define a procedure or function, PCQ Pascal "exports"
the identifier so it can be used by other files. These other
files refer to the routine by including an external reference in
their code (see External References in the Procedures and
Functions section). Thus if you define the procedure DoIt in your
external file, you would include the following declaration in any
other file in which you used the routine:

Procedure DoIt;
External;

That declaration tells the compiler that it should "import" the
identifier from some other file.

Defining variables and typed constants at the global level in an
external file is a bit more complicated. Suppose, for example,
that you have a global variable in your main program, but you also
want to refer to it in your external file. If you declare it
normally in the external file, you might expect that PCQ would
reserve two memory areas for the one variable, one in the main
program and one in the external file.

Because of that problem, variables and typed constants at the
global level (only) are by default assumed to be defined only in
normal program files. Thus if you define a global variable in an
external file, PCQ will assume that the variable should be

imported from some other file, and will not allocate space. This
assumption can be overridden by the $S directive, however.
If you have used an external file, you need to be sure its
routines are included in the final executable program. To do
that, you have to include the file name in the linker command, as
follows:
Blink Main.o Extern1.o ... to MainProgram Library PCQ.lib
In other words you include any external object files (the output
of the assembler) in the linker command after the main file, but
before the "to" keyword. They can be in any order.
Although Pascal is a case-insensitive language, assembly is not.
Thus when you make external references you must be sure that the
very first reference in your code - either the variable
declaration or the actual header for the procedure or function -
has the same case as the definition in the external file.
Subsequent uses can have any case - the first one is used to
generate the reference.

1.24 Notes to assembly programmers

==========
Notes to Assembly Programmers
==========
In previous versions of the compiler, the registers from d2 to d7,
a2 and a3 were all available at any time, and the scratch
registers were available between statements. That is no longer
true, so some assembly routines might have to be rewritten.
Version 1.2 of PCQ now uses registers much more efficiently, which
means that all of the data registers can be put to use. PCQ
allocates registers from d7 down to hold subexpression results, so
the availability of registers within expressions depends on the
Pascal code. Between statements, all data registers are once
again available.
Address registers are used as before: a7 is the stack pointer, a6
is used to point to the library base, a5 points to the stack
frame, and a4 is used to point to previous frames. The compiler
can use a3 and a2 during expressions to hold intermediate address
values.
The system routines consider d0,d1,a0 and a1 as scratch, but
preserve all other registers.

1.25 Errors

==========

Errors

==========

When the compiler runs across some sort of error in your program,
it prints out the current and previous lines, with the general
area of the error highlighted. On the next line, it tells you the
line number and current procedure or function being defined, plus
some (hopefully) descriptive text.
The first error is normally fairly accurate. After that, however,
the compiler might start coming up with a lot of spurious
complaints that are best ignored. For this reason, PCQ Pascal
automatically aborts after four errors (I used to use five, but
apparently the first error was scrolling off of normal screens).
If you specified the "-q" Quiet command line option, the error
reports will have a much more regular form, which is:
"source file name" At ##,## : Error Text
The quotes, colon, and the word "At" are all literal. The first
is the line, and the second is the column. This format is
designed to make automated compilation routines easier to develop.

1.26 Source material

==========

Sources

==========

I wrote PCQ Pascal as a learning experience, and it case you're
wondering I sure learned a lot. Where did I go for information?
First, let's look at information specifically about the Amiga.
Amiga 1.3 Native Developer's Update. This is a package of four
disks distributed by Commodore (CATS, specifically). It has the
complete C and assembly language include libraries, Amiga.lib,
ALink, and all sorts of other things. Perhaps more importantly,
it also has the Amiga Autodocs, a set of documentation files that
explain each and every function available in the standard Amiga
libraries and devices. It explains each one individually,
however, so you still need a more general source. For $20 U.S.,
however, these disks are a bargain. The 2.0 update should be out

eventually, but as I write this it's still unavailable, so you
should send $20 and ask for the 1.3 update from:
C.A.T.S
1200 Wilson Drive
West Chester, PA 19380
Amiga ROM Kernel Manuals. This is the official set of books for
Amiga programming, and if you don't have them you are handicapped.
The last version I have is from AmigaOS version 1.1 so I'm not
sure how they are organized these days, but you should definitely
get the one that explains Intuition, and if you can still afford
it, the one that explains Libraries and Devices. They are big and
very expensive, but it's hard to get by without them. Some folks,
by the way, will also tell you that they are mistake-ridden and
difficult to understand, but I found them to be very accurate and
clear. I think some programmers like to blame their lack of
understanding on others....
Anders Bjerin's C manual. I've never looked at this, so I have no
idea of what it covers. It covers lots of Intuition and Graphics
topics, and the C slant shouldn't be too much of a problem. It's
available on Fred Fish disks 456 & 457, and unpacks to four full
disks of documentation.
As far as information about compilers in particular, I referred to
the following sources:
PDC, a freely distributable C compiler supported by Jeff Lydiatt.
This is a very good program, although it has been eclipsed
recently by DICE, GCC, and other really good C compilers. I
learned, and used, a lot about activation frames from PDC's
output, although PDC's source code remains a mystery.
Pascal-S source code. This is a small demonstration of a Pascal
p-code compiler produced years ago at Wirth's place, ETH Zurich.
It can answer some questions, but isn't a really good example of
programming style.
Small-C source code. Small-C is another freely distributable C
compiler, originally described in Dr. Dobb's Journal years ago.
It is not very powerful, but the simplicity of the source code
makes it a very practical reference. It is one of the compilers
used to bootstrap PCQ.
Brinch Hansen on Pascal Compilers, by Per Brinch Hansen. This
book was of some use, which is more than I can say about the other

half dozen I read while writing this. From this book I mainly learned about all the things I was doing wrong. Great.
The Toy Compiler series in Amiga Transactor, written by Chris Gray. This series is very informative, and is written by the author of Draco. Gray also writes compilers for a living, so he actually knows what he's talking about. Unfortunately this series is hard to find, because the Transactor is out of business. Ask around, though - it's worth it.
Compilers: Principles, Techniques and Tools, by Aho, Sethi and Ullman. This is the big book, the so-called Dragon book, the last word on compiler writing. If you want to know about it, it's in there, but you had better have a pretty decent math background. Although it was a textbook for a class I took, I have never read more than a few pages of it.

1.27 Burner improvements & Update history

==========
Improvements On The Burner
==========
I've been using 1.2 for several months before I issued it, so I'm relatively confident of its quality. Nonetheless, it's such a major change that I expect there will be problems. Therefore my first priority, as always, will be bug fixes.
Next comes optional C calling conventions, integrated peephole optimizing, and possibly IEEE single precision reals. That's all easy enough.
Version 1.3 will be a compatibility release. It will include Turbo-style strings, function return values, double precision reals, and Turbo types: Word, LongInt, ShortInt, etc. Address will become Pointer, Integer will become LongInt, Short will become Integer, Exit will become Halt, etc. I may even implement Unit syntax.
The other area I'd like to explore is debuggers, preferably source level. I've been looking for the source code to one of those monitor programs, but have yet to find it.
==========
Update History
==========

Version 1.2b (March 15, 1992)
o Fixed Write() and Read() of real values, which had
somehow broken yet again.
Version 1.2b (February 15, 1992)
o Fixed the DateTools.i routines
o Fixed a problem with range types with a lower bound in
1..255.
o Comparisons of structured types didn't used to work.
They do now.
o Fixed a problem with the iff.library GetColorTable glue code.
Version 1.2b (September 8, 1991)
o Fixed another problem with array indexing, this time with
multi-dimensional arrays.
o Fixed a mistake in the Exec/ExecBase.i file.
Version 1.2b (August 13, 1991)
o Fixed a problem with array indexing.
o Compiler now accepts empty REPEAT/UNTIL loops again.
o Removed unmatched { in Intuition/IntuitionBase.i
o Modified Ord() to automatically promote single byte
values to the Short type.
Version 1.2b (June 22, 1991)
o The compiler would ignore function or procedure parameters
following negative constants.
o The compiler now ensures that each element of an array is
word-aligned, if the element is larger than one byte.
o Fixed the looping problem with REPEAT/UNTIL.
o The $A didn't work between the routine header and the code
area.
Version 1.2a (June 8, 1991)
Fixed several bugs:
o The XOR operator would produce illegal assembly commands.
o Some floating point comparisons would produce incorrect
results.
o The $A assembly directive now works anywhere in the code,
not just in procedures and functions.
o The compiler used to accept unknown identifiers in typed
constants (as operands of the @ operator).
o The INC and DEC statements were not handling Inc(var1,var2)
forms correctly.

Version 1.2 (April 18, 1991)
Fixed strlen and CreateTask() to work with 32-bit memory.
Re-wrote the expression parsing and code generating routines completely. The routines now use registers much more effectively, and provide a good base for the separate peephole optimizer.
Added a few Turbo Pascal features: you can now use typecasts anywhere, even in address calculations. Thus you can now write something like RecordType(Pointer^).Field, which in version 1.1 was illegal.
Changed the FOR statement significantly in order to make it more efficient, and more like Turbo. It boils down to two differences: FOR loops no longer run a minimum of once (e.g for i := 1 to 0 do ... will execute zero times), and the BY clause is gone.
Added short circuit evaluations, automatic floating point conversions, Heap functions, Reset and Rewrite.
You can now use any standard functions or operators in constant expressions.
Changed the Read routines to comply with Standard Pascal and Turbo, in that any white space is skipped before reading integers and reals. It used to stop at EOLNs.
Version 1.1d, May 6, 1990:
I've begun accumulating small changes under a new version number purely for aesthetics. The first difference is that the compiler now creates a SECTION for data only if it has to. That sounds like an efficiency issue, but actually it wasn't creating a SECTION at all if it was compiling an External unit.
When StdOut is a file, the compiler no longer writes the line numbers and all that stuff. I have also made what I hope is the last fix to the real number reading routines.
The routines that compares include file names to the list of previously included files is now case-insensitive. I can't imagine why it wasn't before.
Version 1.1c, April 6, 1990:
I should have guessed that the problem recognizing the sign of small real numbers on output would have a symmetric problem for input. It did, and now it doesn't. I also fixed the problem with signs on real constants. I also added the exp() and ln() functions recently sent to me by Martin Combs. To avoid inflation

of version numbers, I've changed the date but kept the same version.

I've been thinking about adding peephole optimization to the process, and rather than doing it the right way I've been playing with a separate program. To make that easier, I've removed short branches from the code generated. A68k adds them where necessary anyway.

Version 1.1c, March 3, 1990:

The only changes to the compiler are the new standard functions. The more significant changes were in the runtime library. First, I replaced the sin() and cos() functions based on suggestions by Martin Combs - the result is that the results are accurate to about 3 digits, and only slightly slower. Martin was kind enough to send along a very useful set of routines, which also included the tan() and arctan() functions. I also fixed the routine that writes real numbers, so values between -1.0 and 0.0 now include the minus sign.

Version 1.1b, February 6, 1990:

This program is over a year old.

Added the Sqr() function. Sqr(n) is the same as n * n, but marginally faster and smaller. Also, the compiler used to generate lots of errors when an include file was missing. Now it skips the rest of the comment, like it should.

Apparently floating point constants didn't used to work. Why am I always the last to know? I also added the Sin() and Cos() functions, based on an aside during a lecture on an entirely different topic.

Later I added the sqrt() function, using Newton's method.

Version 1.1a, January 20, 1990:

Fixed a bug in the WriteArb routine that manifested itself whenever you wrote to a 'File of Something'.

Fixed a bug left in the floating point math library. It seems that it had not been updated for the all the 1.1 changes, so during linking it required objects that aren't around anymore. Since floating point math is now handled by the compiler, I hadn't noticed it before.

Version 1.1, December 1, 1989:

This version is completely re-written, and has far too many changes to list them individually here. The main changes are the

with statement, the new IO system, a completely redesigned symbol table, nested procedures, and several new arithmetic operators.
In order to help port programs from Turbo Pascal and C, I added typed constants, the Goto statement, and the normal syntax for multi- dimensional arrays.
Version 1.0c, May 21, 1989:
I changed the input routines around a bit, using DOS files rather than PCQ files. I buffered the input, and made the structure more flexible so I could nest includes. Rather than make up some IfNDef directive, I decided to keep track of the file names included and skip the ones already done. Buffering the input cut compile times in half. I would not have guessed buffering would be that significant, and I suppose I should rethink PCQ input/output in light of this.
I added code to check for the CTRL-C, so you can break out early but cleanly. The Ports.i include file had a couple of errors, which I fixed, and I also fixed the routine that opens a console for programs that need one. It used to have problems when there were several arguments in the first write().
I added the SizeOf() function, floating point math, and the standard functions related to floating point math.
There were several minor problems in the include files which I found when I got the 1.3 includes, the first official set I've had since 1.0.
I relaxed the AND, OR and NOT syntax to allow any ordinal type. This allows you to get bitwise operations on integers and whatever. I also added a standard function called Bit(), described above. These are all temporary until I can get sets into the language.
I finally added string indexing. In doing so I found a bug in the addressing routine selector(), so I rewrote it to be more sensible. I think it also produces larger code, but I'm not too worried because I'm going to add expression trees soon anyway.
Version 1.0b, April 17, 1989:
I fixed a bug in the way complex structures were compared. It seems that one too many bytes were considered, so quite often the comparison would fail.
Version 1.0a, April 8, 1989:
This version added 32 bit math, and fixed the case statement.

The math part was just a matter of getting the proper assembly
source, but I changed the case statement completely. Version 1.0
of the compiler produced a table that was searched sequentially
for the appropriate value, which if found was matched up with an
address. I thought all compilers did this, but when debugging a
Turbo Pascal program at work I found that it just did a bunch of
comparisons before each statement, as if it were doing a series of
optimized if statements. I had thought of this and rejected it as
being too simplistic, but if it's good enough for Turbo it's good
enough for me.
The next thing I changed in this release was the startup code.
You can now run PCQ Pascal programs from the Workbench. This was
just a matter of taking care of the Workbench message, but I also
fooled around with standard input and output. If you try to read
or write to standard in or out from a program launched from the
Workbench, the run time code will open a window for you.
I also fixed one bug that I found: an array index that was
not a numeric type had its type confused. Nevermore.
Version 1.0, February 1, 1989
Original release.

1.28 Other notes & Copyright notice

==========

Other Notes & Copyright

==========

This documentation, the source code for the compiler, the compiler
itself, the source code for the run time library, and the run time
library itself, are all (ahem):

I will allow this version of the package to be freely distributed,
as long as all the files in the archive, with the possible
exception of the assembler and linker (please include them if at
all possible), are included and unchanged. Of course no one can
make any real money for distributing this program. It may only be
distributed on disk collections where a reasonable fee is charged
for the disk itself. A reasonable fee is defined here as the
greater of $10 per disk, or whatever Fred Fish is currently
charging (about six dollars as I write this).

Version 1.2b of PCQ Pascal is "freeware", which means that it can be freely distributed, and no fee is required. Future versions, including the current 1.2c, are now sent only to registered users, and may not be distributed. The registration fee for PCQ Pascal is $25 US, for which you get the following:

o The latest version of the compiler, which now supports variant records, (* and *) delimited comments, CHIP and FAST keywords for global variables and typed constants, standard Pascal pointer declarations, an execution profiler, support for mathtrans.library, and much more.

o PCQ, an extrememly flexible and customizable make utility that lets you compile, assemble and link single programs and complex projects with a single command.

o PCQ.lib.DOC, which describes every routine internal to PCQ Pascal or defined in Include:Utils. Well over 100 routines are covered in a format similar to the AutoDocs, and compatible with DME's REF feature.

o Include files and a runtime library compatible with AmigaDOS 2.0, when they become available.

For an additional $15 (which just covers printing costs, incidentally), I'll send you a printed, bound User's Guide, based on Pascal.DOC, and Reference Guide, based on PCQ.lib.DOC. They are completely reformatted, and even have full indexes.

You can still get the latest version of the freeware compiler from me by sending a disk, mailer and postage. It will only be updated to fix bugs, however, so new versions will not come out very frequently.