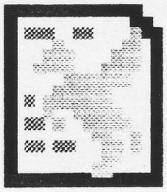
Pegasus Pascal 2.1



Designed and Programmed by Ken O. Burtch.

For questions, comments, or orders, please write to the following address:

Pegasoft
Honsberger Avenue, R.R.#1
Jordan Station, Ontario, Canada
L0R 1S0

Table of Contents

Introduction pg. 1
A. Installation pg. 1
B. Getting Started
Editing Your Programs pg. 2
Compiling Your Programs pg. 2
IBM Pascal Cross-Compiling pg. 2
Linking and Running Your Programs pg. 3
Rebuilding Projects pg. 3
C. Quick Summary of Pegasus Pascal pg. 4
D. Details of Pegasus Pascal 2.1 vs. Standard Pascal
1. Indentation determines the scope of control statements pg. 8
2. PP doesn't have visible statement terminators/separators pg. 8
3. Data declarations pg. 9
3b. Objects and Methods pg. 10
4. Comments pg. 11
5. Literals pg. 11
6. Assigments pg. 11
7. Procedure and Function Calls pg. 11
8. IF ... ELSIF ... ELSE ... pg. 12
9. Loops pg. 12
9b. EXITing loops pg. 13
10. WITH ... pg. 13
11. CASE ... WHEN ... ELSE ... pg. 13
12. GOTO statement pg. 14
13. Built-in Subprograms and Constants pg. 14
14. Expresions pg. 14
15. Subprogram declarations pg. 16
15b. Returning from Subprograms pg. 16
16. Programs pg. 17
17. Units pg. 17
18. Pragmas
i. Conditional Compiling Pragmas pg. 18
ii. Line Macros pg. 18
iii. Output Directives pg. 19
iv. Optimize Directive pg. 19

v. Other Directives pg. 19
vi. ORCA/Pascal Directives pg. 20
19. Reserved Words pg. 22
20. Predefined Identifiers pg. 22
21. Combining Pegasus Pascal with Other ORCA Languages... pg. 22
22. ORCA/Pascal Alternate Symbols / IIGS Font Support pg. 23
E. A Short Tutorial
A Walk-through pg. 24
Debugging pg. 24
A Simple Income Survey pg. 25
A Program to Count Vowels pg. 26
A Program to Find the Square Root of a Real Number pg. 26
A Factorial Program pg. 27
A Summation Program pg. 27
A Program to Bubble-Sort an Integer Array pg. 28
A Unit For Handling Complex Numbers pg. 29
Creating a Large Project that uses Units pg. 31
Linking and Cleanup for Projects pg. 32
F. Troubleshooting pg. 34
G. Glossary pg. 36

Appendix I. PPLib
ExPar: The Expression Parser pg. 37
PPFile: A Faster Way to Manage Files pg. 37
Turtle Graphics Unit pg. 39
Strings Unit pg. 40

Introduction

Pegasus Pascal (or PP) is an advanced computer language for the Apple IIGS and the in-house language of Pegasoft. Based on Pascal, it is designed to reduce spurious syntax errors and offer increased functionality. It requires ORCA/Pascal 1.4 or greater and System 6.0 or greater. Although I have tested it extensively, there is always the possibility of bugs. If you encounter a problem, please send me your source `.pp` file and the `.pas` file generated by PP, with a description of the bug, and I will try to get it fixed.

This manual assumes that you are already familiar with the ORCA/Pascal computer language and the ORCA shell. PP uses many of the same statements found in Pascal, although the format may be different. This manual doesn't contain a Pascal tutorial: there are plenty of books available in local libraries or book stores. Check them out.

For quick starters, read section C.
For a detailed reference, read section D.
For a short tutorial, read section E.

A. Step-by-Step Installation

1. Pegasus Pascal is a shell EXE file for ORCA/Pascal. Place the PP file in the `Utilities` folder of your ORCA disk.

2.To make ORCA aware of this new utility, add the following line to the end of the `SysCmnd` file in the ORCA Shell/System folder:

```
PP                *U                        Pegasus Pascal 2.1
```

3. Replace the contents of your `ORCAPascalDefs` folder with the contents of the `ORCAPascalDefs` folder on your PP disks. These files are up-to-date for System 6.0.1. Note: The names of a couple of toolbox data structures have changed because they conflicted with PP reserved words. Replacing the `ORCAPascalDefs` folder, *in rare cases*, may force you to change existing Pascal programs.

4. Copy the `Icons:PP.Icon` file into the `Icons` folder of your startup disk.

5. There are also help files in the `Help` folder that you can copy into ORCA's `utilities:help` folder. In particular, the file `PP.Errors` contains the pre-compiler errors from Section F.

You are now ready use Pegasus Pascal when you start ORCA.

Problems?
Check the troubleshooting section for more information.

B. Getting Started

Editing Your Programs

There is no language type for a PP source program file. I recommend that you use the PASCAL type and end the name of the source file with a .pp. (The icon file is set up for this format.) Edit the file as you would for any other source file.

A Note on Tabs: Indentation is important in Pegasus Pascal. PP assumes that the editor tabs are the set to the Pascal defaults (every 8 columns). If you have changed your tab settings, keep this in mind.

Compiling Your Programs

Pegasus Pascal is not like other ORCA languages: it's a utility that translates a Pegasus Pascal program into standard Pascal to be compiled by ORCA's compiler. This process is called pre-compiling; PP is a pre-compiler or a "front end" to ORCA/Pascal. Instead of using Compile (or the related commands like CMPLG), you use the PP utility:

```
PP  [+C]  [+E]  [+L]  [+F]  [+M]  [+T]  pathname
```

PP pre-compiles the source file specified by pathname. If there are no errors, ORCA/Pascal will be invoked automatically to finish compiling your program. The following files are produced:

pathname.pp	your Pegasus Pascal program
pathname.pas	the ORCA/Pascal equivalent of your program
pathname.root	the object code for the main program
pathname.a	the object code for the rest of your program

PP can be aborted with ctrl-c (traditional Apple II key combo) or open-apple-period (key combo used by ORCA); it can be paused with the spacebar.

If you specify +L, the Pegasus Pascal produces a listing. The listing can be redirected to a file in the usual way (with > *listingfile*) and printed with a word processor, or can be sent to .printer. A listing line looks like this:

Example: A Listing line

```
32  (9,2)          WriteLn
```

32	the line number
9	the indentation of the line, in spaces
2	declaration nesting level at the start of the line (2 is global declarations)
|	a vertical bar appears if no object code is produced for these lines.

+E will show the expanded version of lines with macros. For example, if Here is a macro for 'Here is a macro' then you would see something like this during a listing:

Example: A Macro Expansion

```
57(9,2)  WriteLn !Here
    ==>  WriteLn 'Here is a macro'
```

The +C option is for a quick syntax check: no pathname.Pas is written to the disk and compiling will be significantly faster. Use this option when you've made many changes to your program and you are think there are a lot of bugs. This only works for individual files, not projects.

IBM Pascal Cross-Compiling

PP supports basic cross-compiling to two popular IBM Pascal's. Pre-compile a file or project with +M to generate Microsoft QuickPascal source code. Pre-compile with +T to generate Turbo Pascal source code. You can copy the ".Pas" file to an IBM machine (for example, with a modem) and compile it. The translated symbols are listed below:

From PP 2.1	To ORCA/Pascal (normal)	To Microsoft Pascal (+M)	To Turbo Pascal (+T)
lines end in C/R	C/R	C/R + L/F	C/R + L/F
BAND	&	AND	AND
BOR	|	OR	OR
BXOR	!	XOR	XOR
BNOT	~	NOT	NOT
<<	<<	SHL	SHL
>>	>>	SHR	SHR
<no inp.,output>	*(input,output)*	*<no input,output>*	*<no input,output>*
EXTERN	EXTERN	EXTERNAL	EXTERNAL
ELSE (case)	OTHERWISE:	ELSE:	ELSE
RANGE ON	{$rangecheck+}	{$R+}	{$R+}
RANGE OFF	{$rangecheck-}	{$R-}	{$R-}
FLOAT ON	{$Float 1}	{$N+}	{$N+}
FLOAT OFF	{$Float 0}	{$N-}	{$N-}

Linking and Running Your Programs

Link and run the object files as you would in any other ORCA language.
For an example of compiling and linking a program, see Section E.

Rebuilding Projects

`PP +F` will fully rebuild a project: the project must be described by a file named `project` in the current directory. PP searches the project list until it finds a program or unit that has been edited. It recompiles that file, and all the rest of the files in the list. If the `+F` is left out, PP only recompiles files that have been edited. This is usually safe, except when changing the export list of a unit, in which case you must use `+F`.

For more information on projects, see section E.

C. Quick Summary of Pegasus Pascal 2.1

For those of you who want to just dive in, here is a summary of Pegasus Pascal. See section D for more details.

Key to symbols:
UPPERCASE: a Pascal keyword - you can't use these words for an identifiers
italics: replace with what the italics refers to; eg. *paragraph* is any legal paragraph
[ ]: denotes an optional part to a command
bold: a comment
...: you continue in the same way

For example, `PRAGMA PExpr [On or Off]` means you can type:
pragma pexpr
pragma pexpr on
pragma pexpr off

Comments, Pragmas and Directives

| - a comment until the end of the line

|$ *ORCA/Pascal directive*

PRAGMA *directive parameters*

Current defined pragmas are:

```
PRAGMA PExpr [On or Off]
PRAGMA Set ident = (or :=) 'string'
PRAGMA If ident = 'string'
PRAGMA Else
PRAGMA Endif
PRAGMA List On or Off
PRAGMA Expand On or Off
PRAGMA Macro ident  ('param' [,'param.'..] )] = (or :=) text
PRAGMA Optimize Space or Off
PRAGMA Range On or Off
PRAGMA Float On or Off
PRAGMA Speak
PRAGMA Source ident
```

Literals

```
'''String'' Literal'        | string literal
$12AB                       | hexidecimal literal
12.34                       | decimal literal
%0101                       | binary literal
{}, {1,2,3}, {a..z}         | set literals
A, A_bc, A2, _A             | identifiers
```

Statements

```
!Macro [ ('param' [,'param2'...] )]          | Macro Expansion
!Compiler Variable
```

```
GOTO ident

ident-label:

IF expr
   paragraph
[ELSIF expr
   paragraph]
[ELSIF ...]
[ELSE
   paragraph]

WHILE expr
    paragraph

REPEAT
   paragraph
UNTIL expr

FOR var = (or :=) expr1 .. expr2 (or expr1 DOWNTO expr2)
    paragraph

WITH var [,var ...]
    paragraph

CASE var
[WHEN label [,Label ...]
   paragraph
[WHEN label ... ]]
ELSE
   paragraph

EXIT [ IF expr
       [ paragraph ] ]

ident[dereference/field/indices] = (or :=) expr
```

Assignment statements may contain &, the reflexive operand.

```
myproc params [& params2 ...]
```

Data Declarations

```
CONST C1 = literal or ident
          [ c2 = literal or ident ... ]
VAR v1 [,v2 ...] : variable type
          [ = initial-value ]
          [ v3 ... ]
TYPE t1 = type definition or object definition
          [t2 = type definition ... ]
LABEL ident1 [, ident2 ...]
```

Built-in Subprograms

```
INC variable [,amount]
DEC variable [,amount]
or any ORCA/Pascal subprograms
```

Subprogram Declarations

```
PROC name-ident [formal_params]
  declarations
BEGIN
  paragraph
END name-ident

FUNC name-ident : type (or name-ident( [formal_params] ) : type)
   declarations
BEGIN
   paragraph
END name-ident

RETURN [IF expr
   [ paragraph ] ]

PROC or FUNC ... EXTERN
PROC or FUNC ... FORWARD
PROC or FUNC ... PRODOS( call-num )
PROC or FUNC ... TOOL(toolset-num, tool-num)
PROC or FUNC ... USERTOOL(toolset-num, tool-num)
PROC or FUNC ... VECTOR(vector-address, tparam)
```

Method Declarations

```
PROC class.method
  declarations
BEGIN
  [INHERITED method]
  paragraph
END class.method

FUNC class.method
   declarations
BEGIN
  [INHERITED method]
  paragraph
END class.method
```

Program Declarations

```
PROGRAM
[[ FROM system or 'path' ] USES ident, ident2, ....
[...] ]
   declarations
BEGIN
   paragraph
END

UNIT
[[ FROM system or 'path' ] USES ident, ident2, ....]
[...]
   exported data declarations & exported subprogram headers
IMPLEMENTATION
   internal data declarations & completed subprograms
END
```

D. Details of Pegasus Pascal 2.1 vs Standard Pascal

1. Indentation determines the scope of control statements

Standard Pascal has a compound statement which is a set of statements starting with the keyword BEGIN and ending the with the keyword END. PP uses paragraphs instead, which are equally indented statements with possible blank lines in between. (Blank lines include lines containing only comments.) Common statements are grouped into paragraphs. Thus, it is extremely important to keep your indentation nicely even. Paragraphs may be used throughout a program.

Consider the IF statement in standard Pascal:

```
if i > 3 then begin
WriteLn('i is > 3');
WriteLn('OK?');
end;
```

The conditional lines must be indented in PP:

```
if i > 3
     WriteLn 'i is > 3'
     WriteLn 'OK?'
```

The indented lines are a paragraph that will be executed if i is greater than 3. The paragraph may contain a blank line:

```
if i > 3
     WriteLn 'i is > 3'

     WriteLn 'OK?'        | This will be executed if i > 3
```

The second WriteLn will be executed the same time as the first one. However, it will be considered OUTSIDE of the IF if it's outdented:

```
if i > 3
     WriteLn 'i is > 3'

WriteLn 'OK?'    | This will always be executed
```

A Strange Case: Consider the IF statement. Because paragraphs use indentation, you can always indent an ELSE (or ELSIF) part more than the IF part, as long as the ELSE is not indented as far as the conditional statements paragraph:

```
if i > 3
   WriteLn 'OK.'
   ELSE | This is weird, but it works
```

BUT if there is an empty conditional paragraph, it isn't legal since the ELSE *becomes* the paragraph:

```
if i > 3
    ELSE     | Else is the conditional - get an error
```

The general rule is to keep statements like IF...ELSIF...ELSE lined up evenly. It looks nicer, and you won't run into this problem.

---

2. PP doesn't have visible statement terminators or separators

PP doesn't use semicolons (;) to end statements; in fact, if you use them, you'll get a warning. Each statement must be on a separate line. If you have a very long line, you can break it up with the hypen symbol (--). The hypen must be the last symbol on the line.

```
x := 3 + (4 + 2) | This is good
x := 3 + ( --
   4 + 2)  | This is good
x := 3 + (
   4 + 2)  | This is BAD
```

This also means you can't use one-line IF's or WHILE's:

```
if x > 3 WriteLn 'wow!' | This must be on 2 separate lines
```

---

3. Data declarations

```
CONST c1 = literal or ident
          [ c2 = literal or ident ... ]
VAR v1 [,v2 ...] : variable type
          [ = initial-value ]
          [ v3 ... ]
TYPE t1 = type definition or object definition
          [t2 = type definition ... ]
LABEL ident1 [, ident2 ...]
```

Data declarations let you define the meaning of the information identifiers in your program. Standard Pascal requires these declarations to occur in the following order: labels, constants, types, and lastly, variables. In PP, these declarations may occur in any order, even interspersed with each other. This way, programmers can group their declarations by use or some other criteria.

```
| Window stuff
Const WindowConst = 25
Var WindowVar : SomeWindowType
Type MyWindows = array[1..WindowConst] of grafportptr

| Menu stuff
Const MenuConst = -5   | no problem in PP!
Var MenuVar : SomeMenuType
```

Records have no END in their definitions, but fields belonging to the record must be in an indented paragraph:

```
Type MyRec = record
          f1 : AField
          f2 : AnotherField
          | No END needed
```

Variant records are similar, except that they resemble PP's CASE statement:

```
Type MyVariantRec = record
          case tag : boolean | tag field is optional
          when true
               f1 : integer
          when false
               f2 : longint
```

Simpe variable declarations can have an optional default value:

```
var Total1, Total2, Total3 : integer
          = 0      | Total1,2,3 start off as zero
```

Variables can also be external:

```
var i : extern integer |i is in my assembly language files
```

Statement labels are identifiers instead of numbers.

```
Label OverHere, OverThere
```

3b. Objects and Methods

```
[OBJECT [( SuperClass )]
      [field-list ("instance-variables") ]
      [...]
      [PROC method-name params or PROC method-name OVERRIDE or
       FUNC method-name(params):type or FUNC method-name OVERRIDE]
      [...]

PROC class.method
  [ declarations ]
BEGIN
  [ INHERITED method ]
  [ statements ]
END class.method

FUNC class.method
  [ declarations ]
BEGIN
  [ INHERITED method ]
  [ statements ]
END class.method
```

Pegasus Pascal offers the same basic object-oriented programming features of ORCA/Pascal. You can declare objects and methods, inherit instance variables from superclasses, etc. Except for the removal of ENDs and semi-colons, the use is the same:

Object Declaration Examples:

```
Type vehicle = OBJECT
       Weight : integer
       Speed  : integer
       x, y   : integer
       proc Move DistanceX, DistanceY : integer
       proc DrawIt
     car = OBJECT( vehicle )
       ManualTransmission : boolean
       Origin : Country
       proc BeepHorn
       proc DrawIt override
```

Method Declarations Examples:

```
proc Vehicle.Move
begin
   x = & + DistanceX
   y = & + DistanceY
end Vehicle.Move

proc Car.DrawIt
  var x1, y1 : integer  | some local variables
begin
```

```
    inherited DrawIt     | execute Vehicle.DrawIt first
    | insert statements to draw a car here
end Car.DrawIt
```

4. Comments

| - a comment until the end of the line

Standard Pascal uses (* ... *) and { ... } to signify comments in a program. As you've seen, PP uses the vertical bar (|) to signify a comment. A comment always runs from a vertical bar to the end of a line, unlike Pascal comments that require a closing symbol. Lines containing only comments do not affect paragraph indentation.

```
if i = 4
    a = b
    | c = d will execute only if i = 4
    c = d
```

5. Literals

Literals are constants that are defined outright in a program. Literals are the same as standard Pascal, except: PP also supports binary literals like %1011. A binary literal starts with a per cent sign (%) and must contain a multiple of 4 binary digits.

Set literals have a minor change: they use curly set brackets ({}) instead of the less-appropriate square brackets ([]).

```
MySet = {a,b,c}
Eight = %1000
```

6. Assignments

ident[*dereference/field/indices*] = (**or** :=) *expr*

PP allows the equal sign (=) or colon-equal (:=) to be used for assignment.

The assignment statement supports a reflexive operand. Something is reflexive if it acts upon itself. If an ampersand (&) appears in the right side of an assignment statement, it will refer to the identifier (without any [],^, etc.) on the left side. Pronounce the ampersand as "itself".

```
i = 5              | same as i := 5
Total = & + 2      | same as Total = Total + 2
List[1] = &[2]     | same as List[1] = List[2]
a = b = c          | same as a := (b = c)
```

7. Procedure and Function Calls

myproc params [& *params2* ...]

Procedure calls have no parentheses (()), resembling the built-in procedures in BASIC. They support a reflexive operator in a similar way to assignments. Using & after a parameter list will re-invoke the procedure with any parameters that follow the &. Pronounce the ampersand as "and again with".

```
LineTo x, y
LineTo x1,y1 & x2,y2 & x3,y3, & x1,y1          | invoke LineTo 4 times
```

Unlike standard Pascal, procedures may not be used as parameters to procedures.

Unlike procedures, function calls require parentheses.

```
MyVar = 1.0 / sin( 0.5 )
```

8. IF expr ... ELSIF expr ... ELSE ...

```
IF expr
   paragraph
[ELSIF expr
   paragraph]
[ELSIF ...]
[ELSE
   paragraph]
```

We've already seen the IF several times. The keyword THEN is omitted and the conditional lines should be in an indented paragraph. PP allows ELSIF parts to an IF, just like Modula and Ada:

```
if i > 3
        a = i
elsif i > 1 | b = i executes only if (i<=3) and (i > 1)
        b = i
else
        WriteLn 'i is less than 2'
```

9. Loops

```
WHILE expr
    paragraph

REPEAT
   paragraph
UNTIL expr

FOR var = (or :=) expr1 .. expr2 (or expr1 DOWNTO expr2)
    paragraph
```

PP uses the same 3 loops as used in Pascal: While (pre-test), Repeat (post-test), and For (iterative). For any loop, the statements to be repeated must be indented.

The REPEAT statement is the same as in standard Pascal:

```
Good:   REPEAT                    Bad: REPEAT
            x = x + 1                   x = x + 1
        UNTIL x >3                  UNTIL x > 3
```

The WHILE statement does not use the keyword DO:

```
while i > j
     i = & - 1
     WriteLn i
```

The FOR statement has undergone several changes. Like an assignment, equals (=) may be used instead of :=. The range is specified as a subrange with an ellipsis (..) instead of the keyword TO. (The keyword DOWNTO works if you want to loop backwards through the range). And like the WHILE, the keyword DO is omitted.

```
for i = 1..5          | same as Pascal's "for i := 1 to 5 do begin"
    WriteLn i
for i := 5 downto 1
    WriteLn i
```

9b. EXITing loops

```
EXIT [ IF expr
       [ paragraph ] ]
```

You can stop repeating any loop with the EXIT statement, which returns you to the next statement after the end of the loop. If you use an IF clause, you will only leave the loop if the clause is true. If you include the optional indented paragraph, the paragraph is executed before you leave the loop.

```
While x > y
     if a > b or c < d      | same as "exit if a > b or c < d"
          exit              | leave the while loop
     x = & + 1

Repeat
     x = & + 1
     exit if StrArray[x] = 'I found it'
          WriteLn 'I found it - now I will the exit the loop'
Until x > StrArraySize
```

10. WITH ptr/record ...

```
WITH var [,var ...]
    paragraph
```

The WITH statement has no DO. The lines to be prefixed should be in an indented paragraph.

```
with rec1, rec2
     WriteLn Field
```

11. CASE ... WHEN ... ELSE ...

```
CASE var
[WHEN label [,Label ...]
   paragraph
[WHEN label ... ]]
ELSE
   paragraph
```

CASEs have undergone several modifications. The keyword OF is omitted. Case labels have no colons; they are proceeded with the keyword WHEN (to hilight the cases). Instead of an optional OTHERWISE: part, PP has a required ELSE part. As usual, all the conditional statements must be in indented paragraphs.

```
case errorMessage
when fatalError
   DisplayBewilderingMessage
   DoHorribleDeathToMakeUserScream
when seriousError, toolboxError
   DoHorribleDeathToMakeUserScream
else
   CrashToSystemMonitor
```

Never indent the WHENs.

12. GOTO Statement

GOTO *statement-ident*
statement-ident:

Statement Labels and GOTO labels are identifers instead of the traditional numbers; if you use numbers, you will get an error message. Because indentation is the key to keeping track of paragraphs, statement labels must always be properly indented to avoid ending paragraphs prematurely.

```
IF x > 3
   WriteLn 'Good'
   Here:
   y = x     | y = x executes if x > 3

 IF x > 3
   WriteLn 'Good'
Here:          | ends  x > 3 paragraph with the WriteLn
   y = x      | y = x always executes
```

The GOTO statement hasn't changed.

```
goto Here
```

13. Built-in Subprograms and Constants

INC *variable* [,*amount*]
DEC *variable* [,*amount*]

PP knows all the built-in procedures, functions and constants of ORCA/Pascal. Remember that procedure calls in PP don't use parantheses. For a list of the predeclared identifiers, see D/20.

```
WriteLn 'WriteLn is a built-in procedure in ORCA/Pascal'
```

PP has two other built-in procedures. INC will add 1 to a variable. DEC will subtract 1 from a variable. The optional expression is the amount you want to inc/dec the variable, if it's other than one.

```
inc x          | same as  x = x + 1 (or x = & + 1)
dec x          | same as  x = x - 1
inc x.r        | same as  x.r = x.r + 1
dec x[2], 3    | same as  x[2] = x[2] - 3
```

14. Expressions

PP uses Modula-2's and BASIC's order of operations (operator precedence): unary operations, brackets, multiplication/division, addition/subtraction, relations (>, =, etc.), then boolean operations (and, or, etc.). Using this order reduces the number of brackets you need in expressions. In general, it's easier to use PP's order-of-ops than standard Pascal's. Consider the statement:

```
if x = 1 and y = 2
```

In standard Pascal, the expression results in a type incompatibility error. Pascal evaluates expressions like this left-to-right. "x = 1 and" is part of a boolean expression, but y is an integer, and you can't AND an integer.

In Pegasus Pascal, AND, OR and XOR are always evaluated LAST. The expression "x = 1" (boolean) can be ANDed with "y = 2" (boolean); there is no error.

Examples:

```
i := 3 * 2 + 4 * 5          | i => (3 * 2) + (4 * 5) => 26
```

```
b := 1 + 2 = 3              | b => (1 + 2) = 3 => true
b := 3 <> 1 + 2             | b => 3 <> (1 + 2) => false
b := 1 > 2 or 3 > 2         | b => (1 > 2) or (3 > 2) => true
b := 1=1 and 2=1 or 3=3     | b => ((1=1) and (2=1)) or (3=3) => true
```

PRAGMA PExpr can force PP to use the standard Pascal order of operations.
The following operators have changed to new keywords/symbols from those used in ORCA/Pascal:

Operation	Pegasus Pascal	ORCA/Pascal
Bitwise NOT	BNOT	~
Bitwise AND	BAND	&
Bitwise OR	BOR	|
Bitwise XOR	BXOR	!
Bit shift left	*no change*	<<
Bit shift right	*no change*	>>
NotEqual	**# or <> or ≠**	<>

Example:
```
x = y BAND %01111111
```

Const statements may have numeric expressions on the right side of the equals sign. The compiler recognizes +,-,*,/ (use / for DIV), (), ** and most ORCA functions. There are a few limitations:

1. The expression may not contain identifiers;
2. The expression cannot start with a function.
3. Numbers cannot be in hex or binary format.
4. Negation will not work properly in all cases (eg. -2+1 will cause an error);
5. Some operators, notably **, only work for integer values.

Example:
```
program
  Const x1  = 1
        x2  = 0.1
        x3  = 0 + abs(1)
        x4  = 0 + arccos(0.5)
        x5  = 0 + arcsin(0.5)
        x6  = 0 + arctan(0.5)
        x7  = 0 + cos(0.5)
        x8  = 0 + exp(0.5)
        x9  = 0 + ln(0.5)
        x10 = 0 + random
        x11 = 0 + randomdouble
        x12 = 0 + randominteger
        x13 = 0 + randomlongint
        x14 = 0 + round(0.5)
        x15 = 0 + sgn(-5)
        x16 = 0 + sin(0.5)
        x17 = 0 + sqr(0.5)
        x18 = 0 + sqrt(9)
        x19 = 0 + tan(0.5)
        x20 = 0 + trunc(1.7)
        x21 = 0 + ord('a')
        x22 = 0 + (3 + 12) / 4
        ScreenPixels = 640 * 200 | number of pixels on 640x200 screen
begin
end
```

15. Subprogram Declarations

```
PROC name-ident [formal_params]
  declarations
BEGIN
  paragraph
END name-ident

FUNC name-ident : type (or name-ident( [formal_params] ) : type)
   declarations
BEGIN
   paragraph
END name-ident

PROC or FUNC ... EXTERN
PROC or FUNC ... FORWARD
PROC or FUNC ... PRODOS( call-num )
PROC or FUNC ... TOOL(toolset-num, tool-num)
PROC or FUNC ... USERTOOL(toolset-num, tool-num)
PROC or FUNC ... VECTOR(vector-address, tparam)
```

PP procedures are declared by the keyword PROC instead of the keyword PROCEDURE (Do you know how many times I've misspelled PROCEDURE when I've been in a hurry?!) and the parameter list has no enclosing parentheses (()). The header must occur on one line unless the hypen is used.

```
proc MyProc   var x,y : integer; j :  ptr
```

PP functions are declared by the keyword FUNC instead of the keyword FUNCTION. The parameter list requires the parentheses.

```
func MyFunc( i : integer ) : boolean
```

Any of ORCA/Pascal's special keywords can be used: forward, tool, prodos or extern, and even univ parameters. They all work the same way as in ORCA/Pascal:

```
proc AnAssemblyLanguageProc extern
```

The executable part of the procedure or function is an indented paragraph enclosed by the keywords BEGIN and END. The keyword END must be followed by the name of the subprogram.

```
proc MyProc
   func MyFunc : integer
   begin
      MyFunc = 0
   end MyFunc
begin
   WriteLn 'Hey! A procedure!'
end MyProc
```

See D/3b on how to declare methods.

15b. Returning from Subprograms

```
RETURN [IF expr
   [ paragraph ] ]
```

The return statement will force the early termination of a procedure or function, much in the way exit works with loops. If the optional if clause is used, the subprogram will only terminate if the clause is true. If the optional

indented paragraph is included, it is executed before the subprogram is terminated.

```
Func SomeMathFunction(x,y : integer) : integer
begin
      return if y = 0
            WriteLn 'Error: can''t divide by 0'
            SomeMathFunction = 0
      SomeMathFunction = x ** 4 div y
end SomeMathFunction
```

16. Programs

```
PROGRAM
[[ FROM system or 'path' ] USES ident, ident2, ....]
[...]
   declarations
BEGIN
   paragraph
END
```

Programs start with the keyword PROGRAM; no name follows, nor any input/output specification. PP always assumes that you'll want to use the the standard INPUT for Read's and OUTPUT for Write's. You need no $Keep directive: PP automatically keeps your object code . A short example program:

```
program
begin
  WriteLn 'This is really short.'
end
```

If you have a USES list, it must come before any declarations and more than one USES is allowed. Here are some examples:

```
Uses MyUnit                 | looks for MyUnit in the current directory
                            | (prefix 8).
From 'misc:' uses x , y     | looks for x and y in the misc subdirectory.
                            | ('...' is not supported yet)
From System uses Common     | same as old "Uses Common".  Use
                            | for toolsets & units in 13:ORCAPascalDefs.
```

17. Units

```
UNIT
[[ FROM system or 'path' ] USES ident, ident2, ....]
[...]
   exported data declarations & exported subprogram headers
IMPLEMENTATION
   internal data declarations & completed subprograms
END
```

Like programs, units start with the keyword UNIT without any name. A $keep is automatically generated for you. The keyword INTERFACE is omitted, and as is the period (.) after the END.

```
unit | a very short unit
implementation
end
```

Compiler variables and line macros cannot be exported from a unit.

Unlike standard Pascal, forward subprograms and subprograms exported from a unit may have

parameters when they are completely defined. This let's you copy and paste a procedure header into a unit's export area without deleting the parameters first. PP checks to make sure all the parameters were declared previously, but it makes no other checks.

Example:

```
unit
proc DrawTriangle x,y : integer
implementation
proc DrawTriangle x,y : integer | "x,y : integer" part is optional
begin
  ...
end DrawTriangle
end
```

Example:

```
program
func CheckStatus : boolean forward
...
func CheckStatus : boolean | ": boolean" part is optional
begin
  ...
end CheckStatus
begin
end
```

---

18. Pragmas

A pragma is a compiler directive. Unlike an ORCA/Pascal directive, a pragma is a statement: it must occur on a line by itself, and it should follow the paragraph indentation rules.

i. Conditional compilation pragmas: Suppose you have a program that contains WriteLn's at strategic places to print out debugging messages. If your program is working, you don't want to compile the messages, but if a bug comes up, you want them there.

Conditional compiling lets you inform the compiler of sections of your program that you may (or may not) want compiled.

In order to conditionally compile, you have to define a compiler variable. PP lets you declare up to 16 variables. You can think of them as a special kind of string variable that only exists while the compiler running. Their scope is from the line where they are declared to the end of the program, and for that reason they usually appear before PROGRAM or UNIT.

```
PRAGMA Set compilerVaraible = 'string'        | declare a compiler variable
```

Example:

```
pragma set UseDebugStatements = 'yes'
pragma set ComputerName = 'Apple IIGS'
```

`PRAGMA If ident = 'string'` Checks the compiler variable to see if it matches string. If it does, the statements following the If directive will be compiled.

`PRAGMA Else` Begins compiling if PP wasn't compiling, or else stops compiling if it was.

`PRAGMA EndIf` End of a conditional compile. Always compiles the statements that follow.

You can't nest conditional compiling statements. A second PRAGMA if will simply override the proceeding conditional compiling PRAGMA.

```
pragma Set  Fruit = 'Oranges'
pragma If   Fruit = 'Oranges'
```

```
WriteLn 'We bought some oranges.'
pragma EndIf | optional if in front of another pragma if
pragma If  Fruit = 'Apples'
WriteLn 'We bought some apples.'
pragma Else
WriteLn 'We didn''t buy apples.'
pragma EndIf
WriteLn 'We went home afterwards.'
```

is the same as

```
WriteLn 'We bought some oranges.'
WriteLn 'We didn''t buy apples.'
WriteLn 'We went home afterwards.'
```

You can test two compiler variables with ! (see below):

```
pragma if x = !y
```

ii. Line Macros: Anyone who has used ORCA/M will know what a macro is. Micol Advanced BASIC users may have used Aliases, which are very similar. Line macros are convenient for short forms of often used statements or pieces of statements. A line macro is a macro limited to a single line of text. When a macro is encountered, the compiler "expands" the macro: unlike a procedure call, the text of the macro is actually inserted into the line in your program to form a new source line.

`PRAGMA MACRO name[('param1' [,'param2'...] ) ] = text` - Defines a new line macro named *name* for the specified *text*. A macro may contain another macro, but remember that macros are only expanded when they are encountered in a source line.

`!name` - Expands the macro or compiler variable, substituting the previously defined text in the source line.

```
program
  pragma macro Hello = 'Hello there!'
  pragma macro ArraySize = 1..100
  Var AnArray : array[ !ArraySize ] of integer
      i : integer
begin
  WriteLn 'This is a test.'
  WriteLn !Hello
  for i = !ArraySize
...
```

is the same as

```
program
  Var AnArray : array[ 1..100 ] of integer
      i : integer
begin
   WriteLn 'This is a test.'
   WriteLn 'Hello there!'
   for i = 1..100
...
```

Line macros can also have parameters. These parameters work in a similar way to macro parameters in the C language. The formal parameters are a list of substrings which can be replaced anywhere in the macro. For example,

```
pragma macro WriteValue('X') = WriteLn 'The value of X is ',X:0,'.'
```

Notice the quote marks around the parameter 'X'. Since X must be a string, only upper case X's are replaced: parameters are case-sensitive. When expanded, every occurance of the letter X in the macro will be relpaced, even if it is in a string literal like 'The value of X is '.

```
       !WriteValue('Total')
-->    WriteLn 'The value of Total is ',Total:0,'.'   | param substring replaced
==>    WRITELN 'The value of Total is ',TOTAL:0,'.'   | resulting line
```

Care must be taken in creating macros for expressions. For example, consider a macro for squaring any expression:

```
pragma macro Square('~x') = ((~x) * (~x))
```

The tilda (~) is not used in Pegasus Pascal. If we had used 'x' instead of '~x', an expansion like

```
y = !Square('x')
```

would cause an infinite number of substitutions. (That is, x's in the string would be replaced with x's, and those x's would be replaced, and so on.) Pegasus Pascal would "time out" after a large number of substitutions and give you an error. This problem will also occur with 'x1' or 'xCoordinate' or 'twoxtwo'. With '~x', the tilda disappears after one replacement and the problem is eliminated.

Also, notice the parentheses. It is usually a good idea if, for macros of expressions, to enclose the whole expression in a set of parentheses as well as enclosing each occurance of the formal parameter. That way, expressions like:

```
       y = !Square('x + y') ** z
-->    ((x + y) * (x + y))
==>    Y = ((X + Y) * (X + Y)) ** Z
```

will evaluate correctly.

iii. Output directives: LISTING is used to turn a compiler listing off or on. EXPAND is used to turn the macro expansion display on or off. These override any +L or +E options used with the PP shell command.

```
pragma listing on
pragma expand off
```

iv. Optimize directive: OPTIMIZE SPACE: Tells the compiler to compress the program as much as possible: literally, it inserts the word PACKED in front of every occurrence of the word SET, FILE, RECORD or ARRAY (if there isn't one already). OPTIMIZE OFF: Stops optimizing. You can further optimize your programs for space and time using the ORCA/Pascal |$optimize directive.

v. Other directives:

`PRAGMA System system_ident` Specifies the source machine/language for a cross-compiler. (eg. Pragma System Apple_IIGS) This pragma is ignored in PP 2.1.

`PRAGMA Range on or off` Turns range checking on or off. This is the same as ORCA/Pascal's $RangeCheck+ and $RangeCheck-. The default is ORCA's default. The pragma is translated when cross-compiling.

`PRAGMA Float On or Off` Turns on or off a coprocessor math board, like Innovative Systems FPE 68881 card (available through Resource-Central). The default is ORCA's default. The pragma is translated when cross-compiling.

`PRAGMA Speak` Starts Byte Work's Talking Tools (if you have them installed) to speak error messages. Once an error message is said, PP switches to your editor without waiting for you to press the Return key (very useful and saves time). You can also impress your friends with your talking computer.

However, there are some bugs: 1. The Talking Tools are not compatible with the Sound Control Panel: the control panel sounds are disabled by the Talking Tools; 2. ORCA's editor becomes slow and misses the occasional

keypress; 3. The Tools don't release their memory properly (perhaps related to point 2).

PRAGMA PExpr [on **or** off] This forces PP to use the older Pascal order of operations. Pragma pexpr off switches back to Pegasus Pascal's order of operations.

Example:

```
pragma pexpr | use standard Pascal expressions
program test
  var x : integer
      s : string[80]
      b : boolean
begin
  pragma pexpr off | use Pegasus Pascal expressions
  x := 5  +  2 * 3
  b := 1 + 2 > 3 or 5 < 4
  pragma pexpr on | use standard Pascal expressions
  s := 'Test'
  WriteLn x,b,s
end
```

In this example, " 11 false Test" is written.

vi. ORCA/Pascal directives can be used by using |$. Most of the ORCA/Pascal directives work fine in Pegasus Pascal, but there are some exceptions.

a. **|$copy and |$append** will not insert a Pegasus Pascal file into your Pegasus Pascal program. (PP ignores these directives, passing them on to ORCA/Pascal, and ORCA/Pascal doesn't know how to compile Pegasus Pascal source code.) You can use |$copy and |$append to insert ORCA/Pascal source code into your Pegasus Pascal program. (For more info on these directives, see your ORCA/Pascal manual.)

b. The following directives **should NOT be used:**
|$Keep - PP generates a |$keep automatically
|$LibPrefix - PP 2.0 generates |$LibPrefix automatically
|$ISO - may work, but PP may use some of the ORCA/Pascal's non-ISO features. For example, PP uses "otherwise:" when it precompiles CASEs.

c. The listing directives **may not work as expected:**
|$Eject - page eject for ORCA/Pascal listing; no effect on PP listings
|$List - generate an ORCA/Pascal listing; no effect on PP listings. Using |$List+ and PRAGMA LIST ON will produce a listing of your Pegasus Pascal source code, followed by a listing of the ORCA/Pascal program PP generated.
|$Title - sets the title for an ORCA/Pascal listing; no effect on PP listings

d. **The following directives will work:**
|$CDev - create a control panel device
|$ClassicDesk - create a classic desk accessory
|$DataBank - save the data bank on subprogram calls
|$Dynamic - begin a dynamic loader segment
|$Float - works properly, but you should use the FLOAT pragma
|$MemoryModel - fast or slow/long addressing
|$Names - save trace info for debugging
|$NBA - create a new botton action
|$NewDeskAcc - create a NDA
|$Optimize - optimize the object code
|$RangeCheck - works properly, but you should use the RANGE pragma
|$RTL - exit program with RTL
|$Segment - begin a static loader segment
|$StackSize - change the stack size
|$ToolParms - use tool calling convention

|$XCMD - create an external command

19. Reserved Words

The following are reserved words in Pegasus Pascal. They cannot be used for identifier names. Also, any ORCA/Pascal reserved words not used by Pegasus Pascal (PROCEDURE, INTERFACE, THEN, DO, TO ...) are also reserved words.

AND	ARRAY	BEGIN	BAND	BNOT	
BOR	BXOR	CASE	CONST	DEC	
DIV	DOWNTO	ELSE	ELSIF	END	
EXIT	EXTERN	FILE	FOR	FORWARD	FROM
FUNC	GOTO	IF	IMPLEMENTATION	IN	
INC	INHERITED	LABEL	MOD	NOT	
OBJECT	OF	OR	OVERRIDE	PACKED	
PRAGMA	PROC	PRODOS	PROGRAM	RECORD	
REPEAT	RETURN	SET	STRING	TOOL	TYPE
UNIT	UNIV	UNTIL	USERTOOL	USES	
VAR	VECTOR	WHEN	WHILE	WITH	

20. Predefined Identifiers

The following are predefined identifiers in Pegasus Pascal. A predefined identifier is a constant, variable, type, procedure or function declared globally by a language. These are available in any Pegasus Pascal program, and can be redefined like any other global identifier; however, it is usually a bad idea to redefine a global identifier, since the original function of the identifier will be lost.

Note: Although INC and DEC are predefined identifiers, PP 2.0 treats them as reserved words.

ABS	ARCCOS	ARCSIN	ARCTAN	ARCTAN2	
BOOLEAN	BYTE	CHR	CHAR	CLOSE	
CNVDS	CNVIS	CNVRS	CNVSD	CNVSI	
CNVSL	CNVSR	COMMANDLINE	CONCAT	COPY	
COS	DEC*	DELETE	DISPOSE	DOUBLE	
ENDDESK	ENDGRAPH	EOF	EOLN	ERROROUTPUT	
EXP	FALSE	GET	HALT	INC*	
INPUT	INSERT	INTEGER	LENGTH	LN	
LONGINT	MAXINT	MAXINT4	MEMBER	NEW	
NIL	ODD	OPEN	ORD	ORD4	
OUTPUT	PACK	PAGE	POINTER	POS	
PRED	PUT	RANDOM	RANDOMDOUBLE	RANDOMINTEGER	
RANDOMLONGINT	READ	READLN	REAL	RESET	
REWRITE	ROUND	ROUND4	SEED	SEEK	
SELF	SHELLID	SIN	SIZEOF	SQR	
SQRT	STARTDESK	STARTGRAPH	SUCC	SYSTEMERROR	
TAN	TEXT	TOOLERROR	TRUE	TRUNC	
TRUNC4	UNPACK	USERID	WRITE	WRITELN	

Example:

```
proc MyProc          | THIS IS NOT RECOMMENDED - JUST AN EXAMPLE.
  type char = boolean     | you can't declare character var's in MyProc
  var i : char            | i is a BOOLEAN - strange, but it is legal
begin
   i:= true
end MyProc
```

21. Combining Pegasus Pascal with Other ORCA Languages: Using EXTERN

In PP 1.0.2, you could treat the Pegasus Pascal programs as if they were ORCA/Pascal programs: you could USE

ORCA/Pascal units in a Pegasus Pascal program, or USE Pegasus Pascal units in an ORCA/Pascal program. Also, you could compile and link ORCA/M and ORCA/C subroutines into Pegasus Pascal programs using EXTERN. EXTERN tells Pegasus Pascal that the Linker will know the files that contain the subroutines. PP accepts the declaration "as is": if the EXTERN declaration is wrong, your program will probably crash when you run it.

In PP 2.1, even though PP translates your programs into ORCA/Pascal, ORCA/Pascal units cannot be included in the USES list (because PP will look for a ".ps" file, and it won't find one). To use ORCA/Pascal units, you will have to use EXTERN, the same way as ORCA/M or ORCA/C.

Example:

```
unit Misc;
  {An ORCA/Pascal unit called "Misc"}
interface uses Common;
  var MiscCount : integer;                          { an exported variable }
  procedure UppercaseString( var s : PString );     { an exported procedure }
implementation
...
end {unit}.

program
  | a Pegasus Pascal program called "main" that uses "misc"
from system uses Common
  var MiscCount : extern integer
  proc UppercaseString var s : PString extern
begin
end
```

To use the unit, use this link command:

```
link main misc keep=main
```

This technique is used in the ExPar demo in the Libraries folder. Since ExPar is an ORCA/Pascal unit, the Parse procedure and the error variable must be external.

22. ORCA/Pascal Alternate Symbols / IIGS Font Support

Standard Pascal allows certain symbols to be used in place of the normal Pascal symbols. In the early days of Pascal, many computers didn't have keyboards with keys for "{", "}" and some of the other keys that are common today. Pegasus Pascal does NOT accept any of these alternate symbols: they are available on the IIGS keyboard.

Also, Pegasus Pascal does not allow EXTERNAL to be used instead of EXTERN.

Pegasus Pascal recognizes the following Apple IIGS font characters in your programs:

π	as 3.141592654	≠	as # or <>
÷	as DIV	≤	as <=
…	as ..	≥	as >=

E. A Short Tutorial on Pegasus Pascal

A Walk-through

Let's go step by step through the creation of your first Pegasus Pascal program. The shortest PP program looks like this:

Pegasus Pascal

```
program
begin
end
```

which, in ORCA/Pascal, is

```
{$keep 'short'}
program short(input,output);
begin
end.
```

This is a very uninteresting program, since it doesn't actually do anything. However, it does reveal the fundamental structure of every PP program:

1. the program header: every program starts with the keyword PROGRAM.
2. the declaration part follows, declaring any variable or subprograms used in the main program. In this case, there isn't any.
3. the keyword BEGIN
4. the executable part follows, the list of instructions the computer is to follow. In this case, there isn't any.
5. the keyword END

You could compile and run this program. PP is very good at doing nothing, but it would be far more interesting if we could perform some practical work. For instance, we could display the answers to some simple calculations on the screen.

```
program
| My first PP program
begin
   WriteLn 2+2 & 3*4

   Write 'The letter after "a" is '
   WriteLn chr( ord( 'a' ) + 1 )
end
```

```
{$keep 'short'}
program short(input,output);
{ My first ORCA/Pascal program }
begin
   WriteLn(2+2);
   WriteLn(3*4);
   Write('The letter after "a" is');
   WriteLn(chr(ord('a')+1));
end.
```

This program contains some instructions for the computer to follow. First, WriteLn is used twice to write the answer to 2+2 and 3*4 on the screen. Second, Write is used to write some text without moving to the next screen line. Finally, WriteLn is used again to display the letter which follows the letter "a". Notice that PP doesn't use parentheses around procedure parameter lists, nor do the statements end with semicolons. The ampersand is used in procedure calls to call a procedure repeatedly. To run this program, type at the ORCA prompt:

1. `Edit short.pp`	to type in the program
2. `pp short.pp`	to compile the program
3. `link short keep=short`	to create an executable program
4. `short`	to run the program

Debugging

The pre-compiler catches all syntax errors (such as spelling keywords wrong); it automatically starts the editor and loads your program for you. Any semantic errors (such as type errors) will be caught by ORCA/Pascal, which automatically loads your PP program. Because the ORCA/Pascal program was generated by the pre-compiler, it's pretty hard to read. To make things easier to debug, the line number of the line in the original Pegasus Pascal program appears in curly set brackets ({}). For instance, in the program "short", you could see something like this from ORCA/Pascal:

```
{7}WRITELN(ORD(ORD('A')+1);
                        ^ type mismatch
```

This means that in line 7 of short.pp, `WriteLn chr( ord( 'a' ) + 1 )`, you typed the word "ord" instead of "chr". Apple-period will abort the compile and switch to the editor. You can now go to line 7 to fix the typo.

PP will check your identifiers to see if they have been declared or not. If PP finds an identifier that is not declared, it checks the list of declared identifiers for a similar identifier. If it only finds one similar identifier, it assumes you made a spelling mistake and corrects the mistake for you and continues compiling.

Example: *If your PP source code is*

```
program
  var Count : integer
begin
  if Coutn = 0
```

PP will give you a warning like this

```
Warning in line n
IF COUTN = 0
   ^---^ Misspelled identifier replaced
------------------------Further Info--------------------
Identifier COUTN is changed to COUNT
```

A Simple Income Survey

The following program performs a simple income survey. It counts the number of people in each income bracket and displays the totals when all the numbers are entered.

```
program
| A simple income survey
  label Start                                  | start of input loop
  var Income : real
  c1, c2, c3, c4, c5 : integer                 | count the number of entries in income
    = 0                                        | categories 1, 2, 3, 4 and 5
  count : integer                              | count the number of people
    = 0
  answer : char                                | used to answer yes/no question
begin
  Start:
  Count = & + 1
  WriteLn 'Enter amount of income for person #',Count:0,':'
  ReadLn Income
  if Income <= 19999.99
    c1 = & + 1
  elsif Income >= 20000 and Income <= 29999.99
    c2 = & + 1
  elsif Income >= 30000 and Income <= 49999.99
    c3 = & + 1
  elsif Income >= 50000 and Income <= 79999.99
    c4 = & + 1
  else
    c5 = & + 1
  WriteLn & 'Enter more data (y/n)?'
  ReadLn Answer
  if Answer = 'Y' or Answer = 'y'
     goto Start
```

```
  WriteLn 'The survey results are'
  WriteLn '-----------------------'
  WriteLn 'Category 1: ',c1
  WriteLn 'Category 2: ',c2
  WriteLn 'Category 3: ',c3
  WriteLn 'Category 4: ',c4
  WriteLn 'Category 5: ',c5
end
```

When this program begin the label start is declared; the variables used in the program are declared, and the totals for the five income categories (c1 to c5) are initialized to 0. The computer repeatedly asks for the income of a person. A category between c1 and c5 is selected by the IF statement. The ELSIF parts are only used if a category has not previously been chosen, and the ELSE part is used if no other category describes the income. If there is more data to enter, the program uses GOTO Start to repeat the instructions starting at the label Start:. The reflexive operand, & (pronounced 'itself') refers the variable being assigned to, in this case, Total.

A Program to Count Vowels

This program counts the number of vowels typed. A person types in one letter at a time, and the program stops when a slash (/) is typed.

```
program
| A Program to count vowels
  var VowelCount : integer                    | count the number of people
    = 0
  ch : char                                   | a letter
begin
  WriteLn 'Please enter your text below (in upper-case),'
  WriteLn 'one letter at a time.  End with ''/''.' &
  ReadLn ch
  while Ch # '/'
    if Ch='A' or Ch='E' or Ch='I' or Ch='O' or Ch='U'
        VowelCount = & + 1
    ReadLn ch
  WriteLn & 'The number of vowels in the text = ',VowelCount:0,'.'
end
```

The variable VowelCount is cleared. The program asks the user to type in one letter at a time. The &, in the WriteLn, calls WriteLn again with no parameters, which leaves a blank like after the message. Instead of using a GOTO, a WHILE is used to repeat the indented statements which follow it. These statements are called a paragraph. The IF and READLN are repeated until a slash is read. After a slash is read, the total is displayed.

A Program to Find the Square Root of a Real Number

This is a program that computes the square root of a real number. You could, of course, use the built-in Pascal function SQRT, but we'll use this to demonstrate the FOR loop.

```
program
| A Program to find the square root of a real number.
  var EPS : real                              | Accuracy
        = 0.000001
      MaxIteration : integer                  | Maximum number of iterations
        = 8
      X0, XOld, N : real
      i : integer
begin
  WriteLn 'Enter the real number:'
```

```
  ReadLn N
  if N >= 0.0
     X0 = N / 2.0 | Just an initial guess
     XOld = X0
     for i = 1..MaxIteration
         exit if abs(X0 * X0 - N) <= EPS * N
         X0 = (& * & + N) / (2.0 * &)
         exit if abs(X0 - XOld) <= (EPS * X0)
         XOld = X0
     WriteLn 'The square root is ',X0
end
```

In this program, the FOR loop will repeat the indented paragraph which follows it MaxIteration times. By changing the constant MaxIteration, you can change the number of times the loop repeats. X0 is the current guess for the value of the square root of N: the more times you repeat, the better the guess will be. EPS is a constant representing the accuracy of the calculation. If a good guess is found before all the iterations have been used, the program jumps out of the FOR loop, using the EXIT IF, and immediately displays the answer.

A Factorial Program

This is a simple program to compute the factorial of a number.

```
program
| A program to find factorials
    var x : integer

    func Fact(n:integer):integer
    begin
        if n > 1
          Fact = n * &( n - 1)
        else
          Fact = 1
    end Fact

begin
    WriteLn 'Give me a number:'
    ReadLn x
    WriteLn 'Factorial is ',Fact(x)
end
```

This program declares a function named Fact. Fact requires an integer parameter (n) and returns an integer value. The & refers the the identifier on the left of the = (the identifier "Fact"). Fact is called a recursive function be it calls itself to compute the factorial of n: The factorial of n equals n * the factorial of n-1. The factorial of 1 or 0 is 1.

A Summation Program

This is a simple summation program.

```
program
    var x : integer

    proc Summation times:integer
        var i, Total : integer
        = 0 | initialize to 0
    begin
        for i = 1 .. times
```

```
        Total = & + i
      WriteLn 'Summation is ',Total
  end Summation
begin
  WriteLn 'Enter a number:'
  ReadLn x
  Summation x
end
```

Summation is a procedure that computes the summation of its parameter, times. The For loop repeats the indented paragraph (that is, the assignment statement) times times. Each time, the value of i is added to Total. The & refers to the identifier to the left of the =, Total.

A Program to Bubble-Sort an Integer Array

A bubble-sort is one of the standard methods of sorting a small quantity of data. The program sorts a list of integer numbers.

```
program
| A Program to bubble-sort an array of integers
  type IArray = array[1..500] of integer
  var  a : IArray
       size, i : integer

proc Bubblesort var a : iarray; var aSize : integer
  var temp, outer_count, i : integer
      b : boolean
begin
  Outer_Count = 1
  repeat
    i = 0
    b = false | b will be true if an swap occurs
    repeat
      inc i
      if a[i] > a[i+1]
         Temp = a[i] | swap a[i] with a[i+1]
         a[i] = a[i+1]
         a[i+1] = temp
         b = true
    until i = aSize - Outer_Count
  until (Outer_Count = aSize) or (not b)
end BubbleSort

begin
  WriteLn 'Enter the size of the array of integers: '
  ReadLn Size

  | We input the array of integers
  for i = 1..Size
      WriteLn 'Enter integer ',i:0,': '
      ReadLn a[i]

  BubbleSort a,Size

  | Output the sorted array
  WriteLn & 'The sorted integers:'
```

```
  WriteLn    '--------------------'
  for i =1..Size
      WriteLn a[i]
end
```

The procedure called Bubblesort has two parameters: a, an integer array, and aSize, the total number of integers in the array. The "Var" in front of "a" means that any changes made to "a" will be reflected in the actual parameter used. The REPEAT loop repeats the indented paragraph between the REPEAT and the UNTIL. INC is a built-in procedure that adds 1 to i: this is the same as i = i + 1, or i = & + 1. Whenever two adjacent integers are out of order, they switch positions in the list. b becomes true if a switch occurs. The process continues until no more swaps occur.

A Unit For Handling Complex Numbers

This is a unit that contains data declarations, procedures, and functions for arithmetic with complex numbers.

```
unit
| A unit that handlee complex numbers.

Type Complex = record      | definition of a complex number
       r         : real    | real part
       imaginary : real    | imaginary part

| Complex input/output

proc ReadComplex var z : Complex
proc WriteComplex z : Complex

| Complex math operations

proc Add z1, z2: Complex; var result : Complex
proc Subtract z1, z2: Complex; var result : Complex
proc Multiply z1, z2: Complex; var result : Complex
proc Divide z1, z2: Complex; var result : Complex
proc Negate z1 : complex; var result : Complex
proc Conjugate z1 : Complex; var result : Complex
proc Absolute z1 : Complex; var result : real
func IsZero( z : complex ) : boolean

implementation

proc ReadComplex var z : Complex
begin
  WriteLn & 'Enter a complex number, real then imaginary,'
  WriteLn 'Ending each with the Return key.' &
  ReadLn z.r & z.imaginary
  WriteLn
end ReadComplex

proc WriteComplex z : Complex
begin
  WriteLn & z.r, ' + i * ',z.imaginary &
end WriteComplex

proc Add z1, z2: Complex; var result : Complex
begin
  result.r = z1.r + z2.r
```

```
  result.imaginary = z1.imaginary + z2.imaginary
end Add

proc Subtract z1, z2: Complex; var result : Complex
begin
  result.r = z1.r - z2.r
  result.imaginary = z1.imaginary - z2.imaginary
end Subtract

proc Multiply z1, z2: Complex; var result : Complex
begin
  result.r = z1.r * z2.r - z1.imaginary * z2.imaginary
  result.imaginary = z1.imaginary * z2.r + z1.r * z2.imaginary
end Multiply

proc Divide z1, z2: Complex; var result : Complex
| Note: no division by zero check
  var Denom : real
begin
  denom = z2.r * z2.r + z2.imaginary * z2.imaginary
  result.r = (z1.r * z2.r + z1.imaginary * z2.imaginary) / denom
  result.imaginary = (z1.imaginary * z2.r - z1.r * z2.imaginary) / denom
end Divide

proc Negate z1 : complex; var result : Complex
begin
  result.r = -z1.r
  result.imaginary = -z1.imaginary
end Negate

proc Conjugate z1 : Complex; var result : Complex
begin
  result.r = z1.r
  result.imaginary = -z1.imaginary
end Conjugate

proc Absolute z1 : Complex; var result : real
begin
  result = sqrt( z1.r * z1.r + z1.imaginary * z1.imaginary )
end Absolute

func IsZero ( z : complex ) : boolean
| IsZero is true if the complex number is close to 0.0
  var result : real
begin
  Absolute z, result
  if result < 0.0001
     IsZero = true
  else
     IsZero = false
end IsZero

end
```

A unit is a collection of procedures, functions, variables, etc. that may be invoked by a program (or programs, or another unit) stored elsewhere in other files. In this unit, Complex is a record containing the real and imaginary

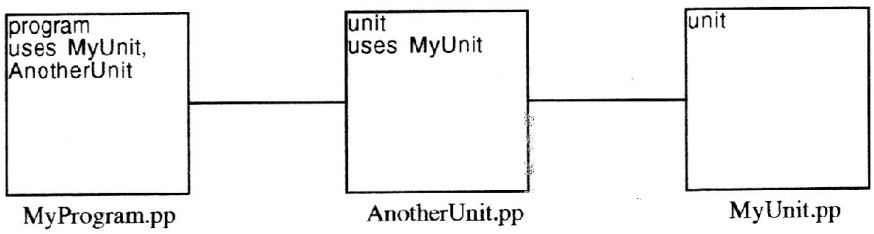
parts of a complex number. Following the data declarations is a list of all procedures and functions in the unit that can be used by other programs. This first part of a unit is called the "interface" or the "export" list. After the keyword `implementation`, the rest of the unit is invisible to outside programs. The implementation part contains the complete declarations of the procedures and functions we described in the interface list.

Creating a Large Project that uses Units

Before starting any large project, you should create a project file. A project file is a text file that contains a list of all the units and the main program you are going to use in your project. When you create a project file, PP can compile all the parts of your project for you. For this reason, the list must be in the order that the units must be compiled, and the main program must always be the last file in the list. (There may be more than one possible arrangement..)

Type `pp +f` with no filename and Pegasus Pascal will search for a file named `project` in the current directory.

Figure 1: A Simple Project



Example:
```
MyUnit.pp
AnotherUnit.pp
MyProgram.pp
```

Pegasus Pascal will read the list, checking the age of each of the files until it finds one that needs compiling. PP then recompiles that file and every file until the end of the list, to make sure the changes haven't affected any of the subsequent units or the main program.

This lets you rebuild a large project safely and easily.

If you type `pp` without the `+F`, PP will recompile only source files that have changed (any files dependant on those source files will be ignored). This rebuilds a project faster, but assumes the unit interface list is unchanged (the identifiers between "Unit" and "Implementation").

In this project, we'll be using one unit and (as always) one program. We'll create a text file, called `Project`, that contains the following:

```
triunit.pp
tridemo.pp
```

If you ever make a program of 1000 lines or more, you'll want to break it up into pieces so you won't have to recompile the whole thing every time you make a tiny change. Here's an example unit (edit it as `triunit.pp`):

```
Unit
    from system uses Common, QuickDrawII

| an example of a simple unit, saved as "TriUnit.pp"
| here's a list of what's defined in the unit

    proc DrawTriangle x,y : integer | a procedure to draw a triangle

implementation| the completed definitions follow

    proc DrawTriangle x,y : integer
    begin
```

```
        MoveTo x,y
        LineTo x,y+30 & x+30,y+15 & x,y
    end DrawTriangle
end
```

DrawTriangle is a procedure to draw a small triangle somewhere on the screen. We might want to use this procedure in a bunch of different programs, so we made a unit and placed it inside. Unlike a program, a unit always starts with the keyword UNIT and ends with the keyword END. Since a unit is only a collection of stuff that's never meant to be executed directly, there is never a main program. Compiling `TriUnit.pp`, you'll get the following files:

```
TriUnit.pp          | our Pegasus Pascal unit
TriUnit.pas         | the translation into ORCA/Pascal
TriUnit.int         | uses list & exported declarations, for the ORCA/Pascal compiler
TriUnit.a           | the actual object code of the unit, for the linker
TriUnit.ps          | exported declarations, for PP
```

Now we'll write a program to use the unit:

```
Program
     from system uses Common, QuickDrawII | list everything TriangleUnit uses
     uses TriUnit                | mention our DrawTriangle unit
begin
    StartGraph 320               | switch to graphic mode
    DrawTriangle 10,10           | draw a triangle at coordinates 10,10
    ReadLn                       | wait until someone presses the return key
    EndGraph                     | switch back to regular, old text mode
end
```

From...Uses tells Pegasus Pascal where the unit files are located. Uses (without a From) tells PP to look in the same directory as the program. If you use From System, PP assumes the necessary files are located in the ORCAPascalDefs folder. Because we're using units, we need a uses list: that's a list of all the units we'll be using. If we saved the DrawTriangle unit in a file with the name `TriUnit.pp`, we include `TriUnit` in our uses list. Then we just use DrawTriangle as if we included it in our program!

To compile our whole project, all we have to type is:

```
pp +f
```

Pegasus Pascal will open our project file and recompile all the files that need to be recompiled. When you link, you'll have to include the names of both the program and the unit.

```
link triunit tridemo keep=tridemo
```

Linking and Cleanup for Projects

If you leave a blank line after the list of files to compile, any additional text is treated as shell commands. LINK is one of the shell commands you can use. You can include here any cleanup commands you want executed when a project has been successfully rebuilt.

Example of a project file:

```
triunit.pp
tridemo.pp

* link the program
link triunit tridemo keep=tridemo
* delete all ".pas" files created by PP
```

```
delete =.pas
* write a message on the screen
echo Running the program ...
* run the program that we compiled and linked
tridemo
```

Because PP executes you shell commands on a one line at a time, don't use loops or branches in your project file.

F. Troubleshooting

File not found :...: If it appears whenever you try to run PP (ie. '...:PP' is not found), check to make sure you installed PP correctly.

The spacing of stuff looks funny in listings: PP ignores tabs in listings. To keep the listings neat, change the fifth default in the SYSTABS file in the shell folder to a 1 instead of a 0. This tells the ORCA editor to translate tabs into the appropriate number of spaces. See your ORCA documentation for more details.

Make/Prizm/EXEC: EGO System's Make and ORCA's Prizm may not work with PP (I get an out-of-memory error). It will work from a standard ORCA EXEC file; I haven't tried the shell version of Make.

Pre-compiler Errors

Actual parameter must be a string literal: Macro parameters must be enclosed in single quote marks.

Binary numbers use groups of four 0's and 1's: This isn't an appropriate binary literal. Make sure you have enough digits.

Can't use defaults here: You can't define default values for global variables in units.

Closing parenthesis missing: Every open parenthesis must have a closed parenthesis. Make sure you have the proper number of each. Also make sure any enumerated types or function parameters are all declared on one line.

Don't know the label *somelabel*: This label hasn't been declared. Check to make sure you spelled it right and that you declared it.

Don't know this Pegasus Pascal directive: There is no such directive for this version of Pegasus Pascal. Make sure you spelled the directive correctly.

Don't recognize this statement: The line does not contain an executable statement. This can happen when ENDs are indented too far.

ELSE without a CASE / IF: An ELSE was found with no corresponding IF or CASE. Check your indentation.

ELSIF without an IF: An ELSIF was found with no corresponding IF. Check your indentation.

End of statement - extra stuff at end of line: The statement was completed at the point indicated. Make sure you don't have too many parentheses.

EXITs too complex: Implementation restriction - you can only use EXIT for the first 9 loops in your (sub)program. Break up the souce into smaller procedures.

EXIT without a loop: An EXIT was found with no corresponding loop. Check your indentation.

Identifier declared (twice): The identifier is already declared at this scope level in the program.

Identifier expected: PP expected an identifier here. Make sure you didn't use any PP reserved words like WHEN or FOR as an identifier.

Identifier not declared: The identifier is not declared at this scope level in the program. Make sure that you spelled the identifier right.

Identifier Possibly Misspelled: The identifier isn't declared, and PP can't find a simple correction.

Infinite substitution into macro: This usually occurs when the formal parameter for a marco in contained in the actual parameter.

Method/Subprogram Expected: A data declaration or statement was found in the subprogram declarations.

Missing quote mark: The closing quote mark appears to be missing, or a string literal is expected.

No DO in Pegasus Pascal: The keyword DO isn't used in PP; PP will usually give you a warning and try to continue compiling as if the keyword wasn't there.

No OF in Pegasus Pascal: The keyword OF isn't used in PP; PP will usually give you a warning and try to continue compiling as if the keyword wasn't there.

No semicolons at end of statements: Semicolons aren't used at the end of a statement; PP will usually give you a warning and try to continue compiling as if the semicolon wasn't there.

No THEN in Pegasus Pascal: The keyword THEN isn't used in PP; PP will usually give you a warning and try to continue compiling as if the keyword wasn't there.

Number expected: A numeric literal is required.

ORCA gave up here: Internal error.

Pre-compiler error *#n*: Internal error.

Proc/Func names don't match: The name given to the subprogram isn't the same as the name given to the END. Check to make sure the subprograms are nested properly and that you spelled the names right in each case.

Syntax error: This is a general error used when there is no other message that covers the condition. This comes up whenever a line doesn't make sense (eg. "for x =..1 5" will cause a syntax error).

There is an unused forward declaration: A procedure or function was declared as FORWARD, or was declared in the interface list of a unit, but was not completed.

Too many closing parentheses: A closing parenthesis was found where it wasn't expected.

Too many labels: The maximum number of labels that PP can handle has been exceeded. Break up your program differently or try using fewer labels.

Too many macros ...: Use less macros in your program.

Too many or too few parameters: Check your parameter list with the macro definition to make sure your parameters are right.

Too many Types/Vars/Consts for me to handle: The maximum number of data declarations that PP can handle has been exceeded. Try to use fewer lines, or break up the declarations differently.

Unexpected ASCII character: An unused character (such as the tilta (~)) appeared in your program.

Use & only in assignments or proc calls: & was used somewhere where it isn't allowed.

WHEN without a CASE: An WHEN was found with no corresponding CASE. Check your indentation.

G. Glossary

Actual Parameter: The variable or value passed to a procedure, function or macro.

Export List: See Interface List.

Formal Parameter: The parameter as declared in a procedure, function or macro header.

Header: For subprograms, the first line of a subprogram, which defines its name and parameters.

Interface List: The declarations in a unit between the "from...uses" lists and the keyword "implementation".

Keyword: A word that has special meaning to a computer language. All PP keywords are reserved words. Unlike Modula-2, keywords are not case-sensitive.

Literal: Any constant that's defined outright in the source code of a program. eg. 123, 'a string', {a,b,c} are all literals.

Macro: In computer languages, one string which stands for another. An alias.

Paragraph: A set of statements that are equally indented equally or more than the first statement.

Pragma: A compiler directive. See D/18 for more details.

Pre-compiler: A 'front-end' to a compiler. Compiles a program into a form that can be compiled by a second compiler.

Project: A collection of units and a main program that composes a complete application, etc.

Reflexive: Acting on oneself; in assignments, an assignment to a variable with an expression that includes the variable itself.

Reserved Word: A keyword that can't be used as an identifier.

Semantics: Loosely, the meaning of a statement. In English, the syntax of the word 'five' is it's spelling, and its semantics is the value 5.

Scope: The range of lines over which an identifier may be used.

Statement: A command for the computer to follow, such as a procedure call or an assignment. Statements must occur on separate lines in PP.

Subprogram: A procedure or function.

Syntax: The structure of a language; see semantics.

Appendix I: PPLib

PPLib (short for "Pegasus Pascal Library") is a collection of useful procedures and functions for your Pegasus Pascal programs. The current library contains four sets of subprograms: ExPar, an expression parser; PPFile, a file manager; Strings, a set of PString handlers; and Turtle, a turtle graphics unit.

To use PPLib in your own programs, please include the following:

Installing PPLib

To install PPLib, simply copy the PPLib file into your Libraries folder, and copy the contents of the accompanying ORCAPascalDefs folder into your Libraries:ORCAPascalDefs folder.

ExPar: The Expression Parser

ExPar evaluates integer (or longint) expressions, useful in simple spreadsheets, databases and financial programs. Since these functions are written in ORCA/Pascal, you will have to declare them as EXTERN in your program. (See Appendix II - Section III, 6.) For an example of what ExPar can do, see the ExPar demo.

There are two simple functions:
1. `func Parse(s:Pstring) : longint`. This evaluates the expression in string s and returns the value as a long integer.
2. `func IsParseError : boolean`. This returns true if an error occurred during the last Parse.

For example,

```
WriteLn Parse('5 * 2 + 10')
```

writes 20 on the screen.

PPFile: A Faster Way to Manage Files

To use PPFile, include Common, GSOS, and PPFile in your FROM SYSTEM USES list.

PPFile is a set of file handling subprograms that allow you to create, destroy and access files. They are similar to the built-in subprograms found in ORCA/Pascal: Open, Reset, etc. PPFile uses GS/OS calls for fast access, and provides you with extra options not found in ORCA/Pascal's subprograms. You cannot interchange ORCA functions with PP functions on the same file. However, because PPFile uses GS/OS calls, you can use GetRefNumGS and use your own GS/OS calls with files opened by PPFile.

Note: When you read from or write to a file, all info is specified by a pointer; the @ operator is an easy way to get a pointer to any variable. For example, in a file of integers, to write a variable "i"'s value to a file, you would use:

```
PPWrite fileID, @i.   | equivalent to ORCA's Write filevar, i.
```

When you open a file, you have to specify the size of a file's records, as well as the size of any initial header you may have created. If you don't have a header (the usual case), make the size 0. Never make the record size 0.

For example, to open and close a file of integers:

```
Const MyFile = 1
...
PPOpen MyFile, 'Int.File', sizeof(integer), 0, 6, 0
| Opens a file named 'Int.File'.
| Int.file contains records the size of integers, and has no header.
| If no file exists, a new one will be created with
```

```
    | filetype 6 (binary file), auxtype 0.
    ...
    PPClose MyFile
    | Closes 'Int.File'
```

PPFile is written in Pegasus Pascal. The unit definition looks like this:

```
unit
  from system uses Common, GSOS

  Const PPMaxFiles = 16
  Type FileID = 1..PPMaxFiles

  | Basic functions

  Proc PPRewrite fid : FileID; fname : PString; recsize, firstsize : integer; --
                 ftype : integer; atype : longint
  Proc PPReset   fid : FileID; fname : PString; recsize, firstsize : integer
  Proc PPOpen    fid : FileID; fname : PString; recsize, firstsize : integer; --
                 ftype : integer; atype : longint
  Proc PPClose   fid : FileID
  Proc PPRead    fid : FileID; data : univ ptr
  Proc PPWrite   fid : FileID; data : univ ptr
  Proc PPRead2   fid : FileID; data : univ ptr; size : longint
  Proc PPWrite2  fid : FileID; data : univ ptr; size : longint

  | Random Access

  Proc PPTop    fid : FileID                    | jump to top of file
  Proc PPBottom fid : FileID                    | jump to bottom of file
  Proc PPSeek   fid : FileID; rec  : longint    | go to a record
  Func PPRec   (fid : FileID) : longint         | current record #
  Func PPSize  (fid : FileID) : longint         | size of file, in records

  | Misc Functions

  Proc PPFlush  fid : FileID    | flushes a file - makes a file safe
  Proc PPCloseAll               | closes all open GS/OS files
  Proc PPFlushAll               | flush all open GS/OS files

  | File Maintenance

  Proc PPDelete fname : PString                 | Deletes the file
  Proc PPRename fname1, fname2 : PString        | Renames file fname1 as fname2
```

PPRewrite opens a new file for writing or overwrites an old one.
PPReset opens an old file for reading.
PPOpen opens a file for reading or writing.
PPClose closes a file. PPCloseAll closes all open (GS/OS) files.
PPRead reads one record and advances to the next position. Records are numbered from 0, as in ORCA.
PPWrite writes one record and advances to the next position.
PPRead2 and PPWrite2 are the same as PPRead and PPWrite, except that they read an arbitrary number of bytes. You can use these to read and write any header info you may store at the start of your file.
PPTop moves to the top of the file. If there is a header, it moves to the header ; otherwise, it moves to record 0.
PPBottom moves to the bottom of the file, the position after the last record. Anything you write will be appended to the file. PPSeek moves to any record. For example, PPSeek fileID, 0 moves to the first record in the file.
PPRec returns the current record number.
PPSize returns the size of the file (in records).

If you leave a file open for a long period of time, you should "flush" the file whenever you update it. Flushing ensures the file is safe, in case of a power failure. PPFlush flushes a file. PPFlushAll flushes all open GS/OS files. In addition, there are two file maintenance functions: PPDelete will delete a file. PPRename will change the name of a file.

Turtle Graphics Unit

Turtle is a turtle graphics library for schools. Turtle graphics was first used in the language Logo, and has since been used to teach math, geometry and computer science to young students. The "turtle" is a triangle that appears in the center of the screen, one point up. It's a marker (cursor) for an imaginary turtle looking towards the top of the screen. When the turtle walks, it draws a line as if a crayon was tied to its tail. The turtle can walk forwards, backwards, turn left or right, and can switch crayons to draw in different colours.

A program called PlayPen, which can be run from the Finder, lets you play with turtle graphics. The menus are designed to reflect the commands you can use from Pegasus Pascal. This program is also a fun introduction to graphics and geometry for young children.

To use turtle graphics in your PP programs, include the Common, QuickDrawII, Turtle names in your FROM SYSTEM USES list. The first two statements of your program must be StartGraph 320 and WakeUp (to wake up the turtle!). Your program should end with EndGraph.

My Turtle library does many of the things that Logo's turtle can do, although some of the names are different. I've tried to keep the names simple and intuitive for children to use. The turtle knows how to do the following things:

```
EraseIt                | erase the screen
MakeIt                 | change the line and scale settings.  You can use it repeatedly.
       _Big            |   increase the drawing scale - zoom in
       _Small          |   decrease the drawing scale - zoom out
       _Fat            |   make lines thicker
       _Thin           |   make lines thinner
       _Normal         |   restore the scale to 100% and lines to 1 pixel thick
Walk steps             | draw a line in the current direction the given steps in length
ColourIt or ColorIt    | change the colour/color.  The turtle knows:
       _Red   _Green _Yellow       _Blue  _Orange
       _Pink  _Brown _Purple       _Violet       _White
       _Black _Grey  _Gray  _Nothing
                       | Use ColourIt _Nothing to move without drawing.
BackUp steps           | draw a line backwards
Left angle             | change the direction left by angle degrees
Right angle            | change the direction right by angle degrees
TurtleOff              | makes the turtle invisible (you will draw faster)
TurtleOn               | makes the turtle visible again
GoHome                 | move the turtle home (to the center screen, facing up)
Print str              | print a string on the screen - eg. Print 'Hello'
SaveIt                 | save the picture as "@:My.Picture", which can be loaded by a
                       | paint program to be edited or printed.
```

Example (of a program using Turtle):

```
program
  from system uses Common, QuickDrawII, Turtle
begin
   StartGraph 320    | switch to 320x200 graphics mode
   WakeUp            | wake up the turtle
   Walk 50           | walk forward 50 pixels
   Right 120         | turn right 120 degrees
   Walk 50           | walk
   Right 120         | turn
   Walk 50           | walk
   Right 120         | turn
   ReadLn            | we drew a triangle - wait for Return key
```

```
        EndGraph          | switch back to text mode
    end
```

Strings Unit

This is a library of useful string handling subprograms. There are character test functions, string conversion procedures, string searches and other useful routines like extra space removal. The test functions recognize foreign characters, although the other procedures do not as yet. Here's a listing of the unit header:

```
unit
from System Uses Common

| String Tests ---------------- |

func IsAlphaNum( s : PString ) : boolean | TRUE if string is alphanumeric
func IsAlpha( s : PString ) : boolean    | TRUE if string is alphabetic
func IsASCII( s : PString ) : boolean    | TRUE if chars are ASCII 0..127
func IsControlC( s : PString ) : boolean | TRUE if string is control chars
func IsInteger( s : PString ) : boolean  | TRUE if string is digits
func IsNumeric( s : PString ) : boolean  | TRUE if string is digits or period
func IsLower( s : PString ) : boolean    | TRUE if string is lower-case
func IsSimilar( s, s2 : PString ) : boolean | case insensitive string test
func IsSpace( s : PString ) : boolean    | TRUE if string is white-space chars
func IsHex( s : PString ) : boolean      | TRUE if string is hex digits
func IsUpper( s : PString ) : boolean    | TRUE if string is upper-case

| String Conversions ----------- |

proc StrToASCII var s : PString | discard high bits
proc StrTOLower var s : PString | convert string to lower case
proc StrToProper var s: PString | convert to proper name / title
proc StrToUpper var s : PString | convert string to upper case
proc StrFix   var s : PString   | remove leading/trailing spaces

| String Processing ------------ |

func CharSearch( s : PString; c : char; p : integer ) : integer
| find the pth occurance of c in s
func StrSearch( s, s2 : PString; p : integer ) : integer
| find the pth occurance of s2 in s
proc StrDuplicate var s : Pstring; times : integer
| duplicate the string s times times
proc StrLeft s : PString; var Left : PString; n : integer
| return the n leftmost characters
proc StrRight s : PString; var Right: PString; n : integer
| return the n rightmost characters
func StrSeed( s : PString ) : integer
| convert s into hash table index value - 0...$3FFF
proc Tab hdist : integer
| move the cursor to a particular column on the text screen
| like Applesoft's tab.
```

Example (of a program using Strings):

```
program
  from system uses Common, Strings
var response : PString
```

```
begin
  WriteLn 'Type "Yes" to stop' &
  repeat
    ReadLn response
  until IsSimilar(Response, 'yes')
end
```