Vector Pascal Reference Manual

Paul Cockshott

21st February 2002

Contents

List of Tables

Chapter 1

Introduction

Vector Pascal is a dialect of Pascal designed to make efficient use of the multi-media instructionsets of recent procesors. It supports data parallel operations and saturated arithmetic. This manual describes the Vector Pascal language.

A number of widely used contemporary processors have instructionset extensions for improved performance in multi-media applications. The aim is to allow operations to proceed on multiple pixels each clock cycle. Such instructionsets have been incorporated both in specialist DSP chips like the Texas C62xx[22] and in general purpose CPU chips like the Intel IA32[10] or the AMD K6 [1].

These instructionset extensions are typically based on the Single Instruction-stream Multiple Data-stream (SIMD) model in which a single instruction causes the same mathematical operation to be carried out on several operands, or pairs of operands at the same time. The level or parallelism supported ranges from 2 floating point operations at a time on the AMD K6 architecture to 16 byte operations at a time on the intel P4 architecture. Whilst processor architectures are moving towards greater levels of parallelism, the most widely used programming languages like C, Java and Delphi are structured around a model of computation in which operations take place on a single value at a time. This was appropriate when processors worked this way, but has become an impediment to programmers seeking to make use of the performance offered by multi-media instructionsets. The introduction of SIMD instruction sets[9][19] to Personal Computers potentially provides substantial performance increases, but the ability of most programmers to harness this performance is held back by two factors. The first is the limited availability of compilers that make effective use of these instructionsets in a machine independent manner. This remains the case despite the research efforts to develop compilers for multi-media instructionsets[6][17][**?**][20]. The second is the fact that most popular programming languages were designed on the word at a time model of the classic von Neumann computer.

Vector Pascal aims to provide an efficient and concise notation for programmers using Multi-Media enhanced CPUs. In doing so it borrows concepts for expressing data parallelism that have a long history, dating back to Iverson's work on APL in the early '60s[13].

Define a vector of type T as having type $T[]$. Then if we have a binary operator $\omega : (T \otimes T) \rightarrow T$, in languages derived from APL we automatically have an operator $\omega : (T[] \otimes T[]) \rightarrow T[]$. Thus if x, y are arrays of integers $k = x + y$ is the array of integers where $k_i = x_i + y_i$.

The basic concept is simple, there are complications to do with the semantics of operations between arrays of different lengths and different dimensions, but Iverson provides a consistent treatment of these. The most recent languages to be built round this model are J, an interpretive language[15][3][16], and F[18] a modernised Fortran. In principle though any language with array types can be extended in a similar way. Iverson's approach to data parallelism is machine independent. It can be implemented using scalar instructions or using the SIMD model. The only difference is speed.

Vector Pascal incorporates Iverson's approach to data parallelism. Its aim is to provide a notation that allows the natural and elegant expression of data parallel algorithms within a base language that is already familiar to a considerable body of programmers and combine this with modern compilation techniques.

By an elegant algorithm I mean one which is expressed as concisely as possible. Elegance is a goal that one approaches asymptotically, approaching but never attaining[5]. APL and J allow the construction of very elegant programs, but at a cost. An inevitable consequence of elegance is the loss of redundancy. APL programs are as concise, or even more concise than conventional mathematical notation[14] and use a special character-set. This makes them hard for the uninitiated to understand. J attempts to remedy this by restricting itself to the ASCII character-set, but still looks dauntingly unfamiliar to programmers brought up on more conventional languages. Both APL and J are interpretive which makes them ill suited to many of the applications for which SIMD speed is required. The aim of Vector Pascal is to provide the conceptual gains of Iverson's notation within a framework familiar to imperative programmers.

Pascal[12]was chosen as a base language over the alternatives of C and Java. C was rejected because notations like `x+y` for `x` and `y` declared as `int x[4],y[4]`, already have the meaning of adding the addresses of the arrays together. Java was rejected because of the difficulty of efficiently transmitting data parallel operations via its intermediate code to a just in time code generator.

Iverson's approach to data parallelism is machine independent. It can be implemented using scalar instructions or using the SIMD model. The only difference is speed. Vector Pascal incorporates Iverson's approach to data parallelism.

Chapter 2

Elements of the language

2.1 Alphabet

In what follows examples and reserved words of Vector Pascal will be denoted in bold face. Vector Pascal programs are made up of letter, digits and special symbols. The digits are shown in table 2.2. The special symbols are shown in table2.3 .

2.2 Reserved words

The reserved words are

ABS, ADDR, AND, ARRAY,
BEGIN, BYTE2PIXEL,
CASE, CDECL, CHR, CONST, COS,
DISPOSE, DIV, DO, DOWNTO,
END, ELSE, EXIT, EXTERNAL,
FALSE, FILE, FOR, FUNCTION,
GOTO,
IF, IMPLEMENTATION, IN, INTERFACE, IOTA,
LABEL, LIBRARY, LN,
MAX, MIN, MOD,
NAME, NDX, NEW, NOT,
OF, OR, ORD,
PACKED, PERM, PIXEL2BYTE, POW,PRED, PROCEDURE, PROGRAM,
RDU, READ, READLN, RECORD, REPEAT, ROUND,
SET, SHL, SHR, SIN, SIZEOF, STRING, SQRT, SUCC,
TAN, THEN, TO, TRANS, TRUE, TYPE,
VAR,
WITH, WHILE, WRITE, WRITELN,
UNIT, UNTIL, USES

Table 2.1: The letters of Vector Pascal.

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z

Table 2.2: The digits of Vector Pascal.

0	1	2	3	4	5	6	7	8	9

Table 2.3: Special symbols

+	:	(
-	'	)
*	=	[
/	<>	]
:=	<	{
.	<=	}
,	>=	^
;	>	..
+:	@	*)
-:	$	(*
_	**	

Reserved words may be written in either lower case or upper case letters, or any combination of the two.

2.3 Comments

The comment construct

{ < any sequence of characters not containing "}" > **}**

may be inserted between any two identifiers, special symbols, numbers or reserved words without altering the semantics or syntactic correctness of the program. The bracketing pair **(* *)** may substitute for **{ }**. Where a comment starts with **{** it continues until the next **}**. Where it starts with **(*** it must be terminated by ***)**[1].

2.4 Identifiers

Identifiers are used to name values, storage locations, programs, program modules, types, procedures and functions. An identifier starts with a letter followed by zero or more letters, digits or the special symbol _. Case is not significant in identifiers.

2.5 Literals

2.5.1 Integer numbers

Integer numbers are formed of a sequence of decimal digits, thus **1**, **23**, **9976** etc, or as hexadecimal numbers, or as numbers of any base between 2 and 36. A hexadecimal number takes the form of a **$** followed by a

[1]Note this differs from ISO Pascal which allows a comment starting with { to terminate with *) and vice versa.

Table 2.4: The hexadecimal digits of Vector Pascal.

Value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Notation 1	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Notation 2											a	b	c	d	e	f

.

sequence of hexadecimal digits thus **$01, $3ff, $5A**. The letters in a hexadecimal number may be upper or lower case and drawn from the range **a..f** or **A..F.**

A based integer is written with the base first followed by a # character and then a sequence of letters or digits. Thus **2#1101** is a binary number **8#67** an octal number and **20#7i** a base 20 number.

The default precision for integers is 32 bits[2].

<digit sequence>	<digit> +

<decimal integer>	<digit sequence>

<hex integer>	'$'<hexdigit>+

<based integer>	<digit sequence>'#'<alphanumeric>+

<unsigned integer>	<decimal integer> <hex integer> <based integer>

2.5.2 Real numbers

Real numbers are supported in floating point notation, thus **14.7, 9.99e5, 38E3, 3.6e-4** are all valid denotations for real numbers. The default precision for real numbers is also 32 bit, though intermediate calculations may use higher precision. The choice of 32 bits as the default precision is influenced by the fact that 32 bit floating point vector operations are well supported in multi-media instructions.

<exp>	'e' 'E'

<scale factor>	[<sign>] <unsigned integer>

<sign>	'-' '+'

[2]The notation used for grammar definition is a tabularised BNF . Each boxed table defines a production, with the production name in the left column. Each line in the right column is an alternative for the production. The metasymbol + indicates one or more repetitions of what immediately preceeds it. The Kleene star * is used for zero or more repetitions. Terminal symbols are in single quotes. Sequences in brackets [ ] are optional.

<unsigned real>	<decimal integer> '.' <digit sequence> <decimal integer>' .' <digit sequence> <exp><scale factor> <decimal integer><exp> <scale factor>

Fixed point numbers

In Vector Pascal pixels are represented as signed fixed point fractions in the range -1.0 to 1.0. Within this range, fixed point literals have the same syntactic form as real numbers.

2.5.3 Character strings

Sequences of characters enclosed by quotes are called literal strings. Literal strings consisting of a single character are constants of the standard type char. If the string is to contain a quote character this quote character must be written twice.

'A' 'x' 'hello' 'John"s house'

are all valid literal strings. The allowable characters in literal strings are:

```
' ' '!'  '"' '#' '$' '%' '&' "' '(' ')' '*' '+' ',' '-' '.'  '/',
'0' '1' '2' '3' '4' '5' '6' '7' '8' '9' ':'  ';' '<' '=' '>' '?',
'@' 'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O',
'P' 'Q' 'R' 'S' 'T' 'U' 'V' 'W' 'X' 'Y' 'Z' '[' '\' ']' '^' '_',
'`' 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o',
'p' 'q' 'r' 's' 't' 'u' 'v' 'w' 'x' 'y' 'z' '{' '|' '}' '~'
```

Chapter 3

Declarations

Vector Pascal is a language supporting nested declaration contexts. A declaration context is either a program context, and unit interface or implementation context, or a procedure or function context. A resolution context determines the meaning of an identifier. Within a resolution context, identifiers can be declared to stand for constants, types, variables, procedures or functions. When an identifier is used, the meaning taken on by the identifier is that given in the closest containing resolution context. Resolution contexts are any declaration context or a **with** statement context. The ordering of these contexts when resolving an identifier is:

1. The declaration context identified by any **with** statements which nest the current occurrence of the identifier. These **with** statement contexts are searched from the innermost to the outermost.
2. The declaration context of the currently nested procedure declarations. These procedure contexts are searched from the innermost to the outermost.
3. The declaration context of the current unit or program.
4. The interface declaration contexts of the units mentioned in the use list of the current unit or program. These contexts are searched from the rightmost unit mentioned in the use list to the leftmost identifier in the use list.
5. The interface declaration context of the System unit.
6. The pre-declared identifiers of the language.

3.1 Constants

A constant definition introduces an identifier as a synonym for a constant.

<constant declaration>	<identifier>=<expression> <identifier>':'<type>'='<typed constant>

Constants can be simple constants or typed constants. A simple constant must be a constant expression whose value is known at compile time. This restricts it to expressions for which all component identifiers are other constants, and for which the permitted operators are given in table3.1 . This restricts simple constants to be of scalar or string types.

Table 3.1: The operators permitted in Vector Pascal constant expressions.

+	-	*	/	div	mod	shr	shl	and	or

Typed constants provide the program with initialised variables which may hold array types.

<typed constant>	<expression> <array constant>

3.1.1 Array constants

Array constants are comma separated lists of constant expressions enclosed by brackets. Thus

tr:array[1..3] of real =(1.0,1.0,2.0);

is a valid array constant declaration, as is:

t2:array[1..2,1..3] of real=((1.0,2.0,4.0),(1.0,3.0,9.0));

The array constant must structurally match the type given to the identifier. That is to say it must match with respect to number of dimensions, length of each dimension, and type of the array elements.

<array constant>	'(' <typed constant> [,<typed constant>]* ')'

3.1.2 Pre-declared constants

maxint The largest supported integer value.

pi A real numbered approximation to π

maxchar The highest character in the character set.

maxstring The maximum number of characters allowed in a string.

maxreal The highest representable real.

minreal The smallest representable positive real number.

epsreal The smallest real number which when added to 1.0 yields a value distinguishable from 1.0.

maxdouble The highest representable double precision real number.

mindouble The smallest representable positive double precision real number.

complexzero A complex number with zero real and imaginary parts.

complexone A complex number with real part 1 and imaginary part 0.

3.2 Labels

Labels are written as digit sequences. Labels must be declared before they are used. They can be used to label the start of a statement and can be the destination of a **goto** statement. A **goto** statement must have as

its destination a label declared within the current innermost declaration context. A statement can be prefixed by a label followed by a colon.

Example

label 99;
begin read(x); if x>9 goto 99; write(x*2);99: end;

3.3 Types

A type declaration determines the set of values that expressions of this type may assume and associates with this set an identifier.

<type>	<simple type> <structured type> <pointer type>

<type definition>	<identifier>'='<type>

3.3.1 Simple types

Simple types are either scalar, standard, subrange or dimensioned types.

<simple type>	<scalar type> <integral type> <subrange type> <dimensioned type> <floating point type>

Scalar types

A scalar type defines an ordered set of identifier by listing these identifiers. The declaration takes the form of a comma separated list of identifiers enclosed by brackets. The identifiers in the list are declared simultaneously with the declared scalar type to be constants of this declared scalar type. Thus

colour = (red,green,blue);
day=(monday,tuesday,wednesday,thursday,friday,saturday,sunday);
are valid scalar type declarations.

Standard types

The following types are provided as standard in Vector Pascal:

integer The numbers are in the range -maxint to +maxint.

real These are a subset of the reals constrained by the IEEE 32 bit floating point format.

double These are a subset of the real numbers constrained by the IEEE 64 bit floating point format.

pixel These are represented as fixed point binary fractions in the range -1.0 to 1.0.

Table 3.2: Categorisation of the standard types.

type	category
real	floating point
double	floating point
byte	integral
pixel	fixed point
shortint	integral
word	integral
integer	integral
cardinal	integral
boolean	scalar
char	scalar

boolean These take on the values **(false,true)** which are ordered such that **true<false**.

char These include the characters from **chr(0)** to **charmax**. All the allowed characters for string literals are in the type char, but the character-set may include other characters whose printable form is country specific.

pchar Defined as **^char**.

byte These take on the positive integers between 0 and 255.

shortint These take on the signed values between -128 and 127.

word These take on the positive integers from 0 to 65535.

cardinal These take on the positive integers form 0 to 4292967295, i.e., the most that can be represented in a 32 bit unsigned number.

longint A 32 bit integer, retained for compatibility with Turbo Pascal.

int64 A 64 bit integer.

complex A complex number with the real and imaginary parts held to 32 bit precision.

Subrange types

A type may be declared as a subrange of another scalar or integer type by indicating the largest and smallest value in the subrange. These values must be constants known at compile time.

<subrange type>	<constant> '..' <constant>

Examples: 1..10, 'a'..'f', monday..thursday.

Pixels

The *conceptual model* of pixels in Vector Pascal is that they are real numbers in the range $-1.0..1.0$. As a signed representation it lends itself to subtraction. As an unbiased representation, it makes the adjustment of

contrast easier. For example, one can reduce contrast 50% simply by multiplying an image by 0.5 [1]. Assignment to pixel variables in Vector Pascal is defined to be saturating - real numbers outside the range $-1..1$ are clipped to it. The multiplications involved in convolution operations fall naturally into place.

The *implementation model* of pixels used in Vector Pascal is of 8 bit signed integers treated as fixed point binary fractions. All the conversions necessary to preserve the monotonicity of addition, the range of multiplication etc, are delegated to the code generator which, where possible, will implement the semantics using efficient, saturated multi-media arithmetic instructions.

Dimensioned types

These provide a means by which floating point types can be specialised to represent dimensioned numbers as is required in physics calculations. For example:

kms =(mass,distance,time);
meter=real of distance;
kilo=real of mass;
second=real of time;
newton=real of mass * distance * time POW -2;
meterpersecond = real of distance *time POW -1;
The grammar is given by:

<dimensioned type>	<real type> <dimension >['*' <dimension>]*

<real type>	'real' 'double'

<dimension>	<identifier> ['POW' [<sign>] <unsigned integer>]

The identifier must be a member of a scalar type, and that scalar type is then referred to as the basis space of the dimensioned type. The identifiers of the basis space are referred to as the dimensions of the dimensioned type. Associated with each dimension of a dimensioned type there is an integer number referred to as the power of that dimension. This is either introduced explicitly at type declaration time, or determined implicitly for the dimensional type of expressions.

A value of a dimensioned type is a dimensioned value. Let $\log_d t$ of a dimensioned type t be the power to which the dimension d of type t is raised. Thus for t =newton in the example above, and d =time, $\log_d t = -2$

If x and y are values of dimensioned types $t_x$and $t_y$respectively, then the following operators are only permissible if $t_x = t_y$

+	-	<	>	<>	=	<=	>=

For + and -, the dimensional type of the result is the same as that of the arguments. The operations

*	/

[1]When pixels are represented as integers in the range 0..255, a 50% contrast reduction has to be expressed as $((p - 128) \div 2) + 128$.

are permitted if the types $t_x$and t_y share the same basis space, or if the basis space of one of the types is a subrange of the basis space of the other.

The operation **POW** is permitted between dimensioned types and integers.

Dimension deduction rules

1. If $x = y * z$ for $x : t_1, y : t_2, z : t_3$ with basis space B then

   $\forall_{d \in B} \log_d t_1 = \log_d t_2 + \log_d t_3$.

2. If $x = y/z$ for $x : t_1, y : t_2, z : t_3$ with basis space B then

   $\forall_{d \in B} \log_d t_1 = \log_d t_2 - \log_d t_3$.

3. If $x = y$ **POW** z for $x : t_1, y : t_2, z : integer$ with basis space for t_2, B then

   $\forall_{d \in B} \log_d t_1 = \log_d t_2 \times z$.

3.3.2 Structured types

Static Array types

An array type is a structure consisting of a fixed number of elements all of which are the same type. The type of the elements is referred to as the element type. The elements of an array value are indicated by bracketed indexing expressions. The definition of an array type simultaneously defines the permitted type of indexing expression and the element type.

The index type of a static array must be a scalar or subrange type. This implies that the bounds of a static array are known at compile time.

<array type>	'array' '[' <index type>[,<index type>]* ']' 'of' <type>

<index type>	<subrange type> <scalar type> <integral type>

Examples
array[colour] of boolean;
array[1..100] of integer;
array[1..2,4..6] of byte;
array[1..2] of array[4..6] of byte;

The notation [*b,c*] in an array declaration is shorthand for the notation [*b*]**of array**[*c*]. The number of dimensions of an array type is referred to as its rank. Scalar types have rank 0.

String types

A string type denotes the set of all sequences of characters up to some finite length and must have the syntactic form:

<string-type>	'string[' <integer constant>']' 'string'

the integer constant indicates the maximum number of characters that may be held in the string type. The maximum number of characters that can be held in any string is indicated by the pre-declared constant **maxstring**. The type **string** is shorthand for **string[maxstring]**.

Record types

A record type defines a set of similar data structures. Each member of this set, a record instance, is a Cartesian product of number of components or *fields* specified in the record type definition. Each field has an identifier and a type. The scope of these identifiers is the record itself.

A record type may have as a final component a *variant part*. The variant part, if a variant part exists, is a union of several variants, each of which may itself be a Cartesian product of a set of fields. If a variant part exists there may be a tag field whose value indicates which variant is assumed by the record instance.

All field identifiers even if they occur within different variant parts, must be unique within the record type.

<record type>	'record' <field list> 'end'

<field list>	<fixed part> <fixed part>';' <variant part> <variant part>

<fixed part>	<record section> [';' <record section.]*

<record section>	<identifier>[',' <identifier>]* ':' <type> <empty>

<variant part>	'case' [<tag field> ':'] <type identifier> 'of'<variant>[';' <variant>]*

<variant>	<constant> [',' <constant>]*':' '(' <field list> ')' <empty>

Set types

A set type defines the range of values which is the power-set of its base type. The base type must be a scalar type, a character type, integer type or a subrange thereof.

<set type>	'set' 'of' <base type>

3.3.3 Dynamic types

Variables declared within the program are accessed by their identifier. These variables exist throughout the existence of the scope within which they are declared, be this unit, program or procedure. These variables are assigned storage locations whose addresses, either absolute or relative to

some register, can be determined at compile time. Such locations a referred to as static[2]. Storage locations may also be allocated dynamically. Given a type **t**, the type of a pointer to an instance of type **t** is **^t**.

A pointer of type **^t** can be initialised to point to a new store location of type t by use of the built in procedure **new**. Thus if **p:^t**,

new(p);

causes **p** to point at a store location of type **t**.

Pointers to dynamic arrays

The types pointed to by pointer types can be any of the types mentioned so far, that is to say, any of the types allowed for static variables. In addition however, pointer types can be declared to point at dynamic arrays. A dynamic array is an array whose bounds are determined at run time.

Pascal 90[11] introduced the notion of schematic or parameterised types as a means of creating dynamic arrays. Thus where **r** is some integral or ordinal type one can write

type z(a,b:r)=array[a..b] of t;

If **p:^z**, then

new(p,n,m)

where **n,m:r** initialises **p** to point to an array of bounds **n..m**. The bounds of the array can then be accessed as **p^.a, p^.b**. Vector Pascal currently allows dynamic but not static parameterised types.

3.4 File types

A type may be declared to be a file of a type. This form of definition is kept only for backward compatibility. All file types are treated as being equivalent. A file type corresponds to a handle to an operating system file. A file variable must be associated with the operating system file by using the procedures **assign, rewrite, append**, and **reset** provided by the system unit. A pre-declared file type **text** exists.

3.5 Variables

Variable declarations consist of a list of identifiers denoting the new variables, followed by their types.

<variable declaration>	<identifier> [',' <identifier>]* ':' <type><extmod>

Variables are abstractions over values. They can be either simple identifiers, components or ranges of components of arrays, fields of records or referenced dynamic variables.

<variable>	<identifier>
	<indexed variable>
	<indexed range>
	<field designator>
	<referenced variable>

[2]The Pascal concept of static variables should not be equated with the notion of static variables in some other languages such as C or Java. In Pascal a variable is considered static if its offset either relative to the stack base or relative to the start of the global segment can be determined at compile/link time. In C a variable is static only if its location relative to the start of the global segment is known at compile time.

Examples
x,y:real;
i:integer;
point:^real;
dataset:array[1..n]of integer;
twoDdata:array[1..n,4..7] of real;

3.5.1 External Variables

A variable may be declared to be external by appending the external modifier.

<extmod>	';' 'external' 'name' <stringlit>

This indicates that the variable is declared in a non Vector Pascal external library. The name by which the variable is known in the external library is specified in a string literal.
Example
count:integer; external name '_count';

3.5.2 Entire Variables

An entire variable is denoted by its identifier. Examples **x,y,point**,

3.5.3 Indexed Variables

A component of an *n* dimensional array variable is denoted by the variable followed by *n* index expressions in brackets.

<indexed variable>	<variable>'[' <expression>[','<expression>]* ']'

The type of the indexing expression must conform to the index type of the array variable. The type of the indexed variable is the component type of the array.
Examples
twoDdata[2,6]
dataset[i]
Given the declaration
a=array[p] of q
then the elements of arrays of type **a**, will have type **q** and will be identified by indices of type **p** thus:
b[i]
where **i:p**, **b:a**.
Given the declaration
z = string[x]
for some integer x $\leq$**maxstring**, then the characters within strings of type **z** will be identified by indices in the range **1..x,** thus:
y[j]
where **y:z**, **j:1..x**.

3.5.4 Indexed Ranges

A range of components of an array variable are denoted by the variable followed by a range expression in brackets.

<indexed range>	<variable> '[' <range expression>[',' <range expression>]* ']'

<range expression>	<expression> '..' <expression>

The expressions within the range expression must conform to the index type of the array variable. The type of a range expression **a[i..j]** where **a: array[p..q] of t** is **array[0..j-i] of t.**

Examples:

dataset[i..i+2]:=blank;

twoDdata[2..3,5..6]:=twoDdata[4..5,11..12]*0.5;

Subranges may be passed in as actual parameters to procedures whose corresponding formal parameters are declared as variables of a schematic type. Hence given the following declarations:

type image(miny,maxy,minx,maxx:integer)=array[miny..maxy,minx..maxx] of byte;

procedure invert(var im:image);begin im:=255-im; end;

var screen:array[0..319,0..199] of byte;

then the following statement would be valid:

invert(screen[40..60,20..30]);

3.5.5 Virtual array variables

If an array variable occurs on the right hand side of an assignment statement, there is a further form of indexing possible. An array may be indexed by another array. If **x:array[t0] of t1** and **y:array[t1] of t2**, then **y[x]** denotes the virtual array of type **array[t0] of t2** such that **y[x][i]=y[x[i]]**. This construct is useful for performing permutations. To fully understand the following example refer to sections 4.1.3,4.2.1.

Example Given the declarations

const perm:array[0..3] of integer=(3,1,2,0);

var ma,m0:array[0..3] of integer;

then the statements

m0:= (iota 0)+1;

write('m0=');for j:=0 to 3 do write(m0[j]);writeln;

ma:=m0[perm];

write('perm=');for j:=0 to 3 do write(perm[j]);writeln;

writeln('ma:=m0[perm]');for j:=0 to 3 do write(ma[j]);writeln;

would produce the output

```
m0= 1 2 3 4
perm=  3 1 2 0
ma:=m0[perm]
4 2 3 1
```

3.5.6 Field Designators

A component of an instance of a record type, or the parameters of an instance of a schematic type are denoted by the record or schematic type instance followed by the field or parameter name.

<field designator>	<variable>'.'<identifier>

3.5.7 Referenced Variables

If **p:^t**, then **p^** denotes the dynamic variable of type **t** referenced by **p**.

<referenced variable>	<variable> '^'

3.6 Procedures and Functions

Procedure and function declarations allow algorithms to be identified by name and have arguments associated with them so that they may be invoked by procedure statements or function calls.

<procedure declaration>	<procedure heading>';'[<proc tail>]

<proc tail>	'forward'	must be followed by definition of procedure body
	'external'	imports a non Pascal procedure
	<block>	procedure implemented here

<paramlist>	'('<formal parameter section>[';'<formal parameter section>]*')'

<procedure heading>	'procedure' <identifier> [<paramlist>] 'function'<identifier> [<paramlist>]':'<type>

<formal parameter section>	['var']<identifier>[','<identifier>]':'<type>

The parameters declared in the procedure heading are local to the scope of the procedure. The parameters in the procedure heading are termed formal parameters. If the identifiers in a formal parameter section are preceded by the word **var**, then the formal parameters are termed variable parameters. The block[3] of a procedure or function constitutes a scope local to its executable compound statement. Within a function declaration there must be at least one statement assigning a value to the function identifier. This assignment determines the result of a function, but assignment to this identifier does not cause an immediate return from the function.

Function return values can be scalars, pointers, records, strings or sets. Arrays may not be returned from a function.

Examples The function sba is the mirror image of the abs function.

```
function sba(i:integer):integer;
begin if i>o then sba:=-i else sba:=i end;
type stack:array[0..100] of integer;
procedure push(var s:stack;i:integer);
begin s[s[0]]:=i;s[0]:=s[0]+1; end;
```

[3] see section 5.

Chapter 4

Algorithms

4.1 Expressions

An expression is a rule for computing a value by the application of operators and functions to other values. These operators can be *monadic* - taking a single argument, or *dyadic* - taking two arguments.

4.1.1 Mixed type expressions

The arithmetic operators are defined over the base types integer and real. If a dyadic operator that can take either real or integer arguments is applied to arguments one of which is an integer and the other a real, the integer argument is first implicitly converted to a real before the operator is applied. Similarly, if a dyadic operator is applied to two integral numbers of different precision, the number of lower precision is initially converted to the higher precisions, and the result is of the higher precision. Higher precision of types *t,u* is defined such that the type with the greater precision is the one which can represent the largest range of numbers. Hence reals are taken to be higher precision than longints even though the number of significant bits in a real may be less than in a longint.

When performing mixed type arithmetic between pixels and another numeric data type, the values of both types are converted to reals before the arithmetic is performed. If the result of such a mixed type expression is subsequently assigned to a pixel variable, all values greater than 1.0 are mapped to 1.0 and all values below -1.0 are mapped to -1.0.

4.1.2 Primary expressions

<primary expression>	'(' <expression> ')'
	<literal string>
	'true'
	'false'
	<unsigned integer>
	<unsigned real>
	<variable>
	<constant id>
	<function call>
	<set construction>

The most primitive expressions are instances of the literals defined in the language: literal strings, boolean literals, literal reals and literal integers. 'Salerno', **true**, 12, $ea8f, 1.2e9 are all primary expressions. The next level of abstraction is provided by symbolic identifiers for values. **X**, **left**, **a.max**, **p^.next**, **z[1]**, **image[4..200,100..150]** are all primary expressions provided that the identifiers have been declared as variables or constants.

An expression surrounded by brackets **( )** is also a primary expression. Thus if *e* is an expression so is **(** *e* **)**.

<function call>	<function id> ['(' <expression> [,<expression>]* ')']

<element>	<expression> <range expression>

Let *e* be an expression of type t_1 and if **f** is an identifier of type **function(**t_1 **):**t_2, then **f(** *e* **)** is a primary expression of type t_2. A function which takes no parameters is invoked without following its identifier by brackets. It will be an error if any of the actual parameters supplied to a function are incompatible with the formal parameters declared for the function.

<set construction>	'[' [<element>[,<element>]*] ']'

Finally a primary expression may be a set construction. A set construction is written as a sequence of zero or more elements enclosed in brackets **[ ]** and separated by commas. The elements themselves are either expressions evaluating to single values or range expressions denoting a sequence of consecutive values. The type of a set construction is deduced by the compiler from the context in which it occurs. A set construction occurring on the right hand side of an assignment inherits the type of the variable to which it is being assigned. The following are all valid set constructions:

[], [1..9], [z..j,9], [a,b,c,]

[] denotes the empty set.

4.1.3 Unary expressions

A unary expression is formed by applying a unary operator to another unary or primary expression. The unary operators supported are **+, -, *, /, div, mod, and, or, not, round, sqrt, sin, cos, tan, abs, ln, ord, chr, byte2pixel, pixel2byte, succ, pred, iota, trans, addr** and **@**.

Thus the following are valid unary expressions**: -1, +b, not true, sqrt abs x, sin theta.** In standard Pascal some of these operators are treated as functions,. Syntactically this means that their arguments must be enclosed in brackets, as in **sin(theta)**. This usage remains syntactically correct in Vector Pascal.

The dyadic operators **+, -, *, /, div, mod , and or** are all extended to unary context by the insertion of an implicit value under the operation. Thus just as **-a = 0-a** so too **/2 = 1/2**. For sets the notation **-s** means the complement of the set **s**. The implicit value inserted are given below.

type	operator**s**	**implicit value**
number	**+,-**	0
string	**+**	”
set	**+**	empty set
set	**-,***	full-set
number	***,/ ,div,mod**	1
number	**max**	lowest representable number of the type
number	**min**	highest representable number of the type
boolean	**and**	true
boolean	**or**	false

A unary operator can be applied to an array argument and returns an array result. Similarly any user declared function over a scalar type can be applied to an array type and return an array. If **f** is a function or unary operator mapping from type **r** to type **t** then if **x** is an array of **r,** and **a** an array of **t**, then **a:=f(x)** assigns an array of **t** such that **a[i]=f(x[i])**

lhs	rhs	meaning
<unaryop>	’+’	+x = 0+x identity operator
	’-’	-x = 0-x,
		note: this is defined on integer, real and complex
	’*’	*x=1*x identity operator
	’/’	/x=1.0/x
		note: this is defined on integer, real and complex
	’div’	div x =1 div x
	’mod’	mod x = 1 mod x
	’and’	and x = true and x
	’or’	or x = false or x
	’not’	complements booleans
	’round’	rounds a real to the closest integer
	’sqrt’	returns square root as a real number.
	’sin’	sine of its argument. Argument in radians. Result is real.
	’cos’	cosine of its argument. Argument in radians. Result is real.
	’tan’	tangent of its argument. Argument in radians. Result is real.
	’abs’	if x<0 then abs x = -x else abs x= x
	’ln’	$\log_e$ of its argument. Result is real.
	’ord’	argument scalar type, returns ordinal
		number of the argument.
	’chr’	converts an integer into a character.
	’succ’	argument scalar type,
		returns the next scalar in the type.
	’pred’	argument scalar type,
		returns the previous scalar in the type.
	’iota’	iota i returns the ith current index
	’trans’	transposes a matrix or vector
	’pixel2byte’	convert pixel in range -1.0..1.0 to byte in range 0..255
	’byte2pixel’	convert a byte in range 0..255 to a pixel in
		the range -1.0..1.0
	’@’,’addr’	Given a variable, this returns an
		untyped pointer to the variable.

<unary expression>	<unaryop> <unary expression> 'sizeof' '(' <type> ')' <operator reduction> <primary expression>
	'if'<expression> 'then' <expression> 'else' <expression>

sizeof

The construct **sizeof(** *t* **)** where *t* is a type, returns the number of bytes occupied by an instance of the type.

iota

The operator iota i returns the ith current implicit index[1].

Examples Thus given the definitions

var v1:array[1..3]of integer;
v2:array[0..4] of integer;
then the program fragment
v1:=iota 0;
v2:=iota 0 *2;

for i:=1 to 3 do write(v1[i]); writeln;
writeln('v2');
for i:=0 to 4 do write(v2[i]); writeln;
would produce the output

```
v1
1 2 3
v2
0 2 4 6 8
```

whilst given the definitions

m1:array[1..3,0..4] of integer;m2:array[0..4,1..3]of integer;
then the program fragment
m2:= iota 0 +2*iota 1;
writeln('m2:= iota 0 +2*iota 1 ');
for i:=0 to 4 do begin for j:=1 to 3 do write(m2[i,j]); writeln; end;
would produce the output

```
m2:= iota 0 +2*iota 1
2 4 6
3 5 7
4 6 8
5 7 9
6 8 10
```

The argument to **iota** must be an integer known at compile time within the range of implicit indices in the current context. The reserved word **ndx** is a synonym for **iota**.

[1]See section 4.2.1.

perm A generalised permutation of the implicit indices is performed using the syntactic form:

perm[*index-sel[,index-sel]*]*expression*

The *index-sel*s are integers known at compile time which specify a permutation on the implicit indices. Thus in e evaluated in context **perm**$[i,j,k]e$, then:

iota 0 = iota i, **iota 1= iota** j, **iota 2= iota** k

This is particularly useful in converting between different image formats. Hardware frame buffers typically represent images with the pixels in the red, green, blue, and alpha channels adjacent in memory. For image processing it is convenient to hold them in distinct planes. The **perm** operator provides a concise notation for translation between these formats:

```
type rowindex=0..479;
     colindex=0..639;
var channel=red..alpha;
    screen:array[rowindex,colindex,channel] of pixel;
    img:array[channel,colindex,rowindex] of pixel;
...
screen:=perm[2,0,1]img;
```

trans and **diag** provide shorthand notions for expressions in terms of **perm**. Thus in an assignment context of rank 2, **trans = perm[1,0]** and **diag = perm[0,0]**.

trans

The operator trans transposes a vector or matrix. It achieves this by cyclic rotation of the implicit indices. Thus if **trans** *e* is evaluated in a context with implicit indices
iota *0..* **iota** n
then the expression e is evaluated in a context with implicit indices
iota'*0..* **iota**'n
where
iota'x = **iota** ((x+*1*)**mod** n+*1*)
It should be noted that transposition is generalised to arrays of rank greater than 2.

Examples Given the definitions used above in section 4.1.3, the program fragment:

```
m1:= (trans v1)*v2;
writeln('(trans v1)*v2');
for i:=1 to 3 do begin for j:=0 to 4 do write(m1[i,j]); writeln; end;
m2 := trans m1;
writeln('transpose 1..3,0..4 matrix');
for i:=0 to 4 do begin for j:=1 to 3 do write(m2[i,j]); writeln; end;
```

will produce the output:

```
(trans v1)*v2
0  2  4  6  8
0  4  8 12 16
```

```
0  6 12 18 24
transpose 1..3,0..4 matrix
0  0  0
2  4  6
4  8 12
6 12 18
8 16 24
```

4.1.4 Operator Reduction

Any dyadic operator can be converted to a monadic reduction operator by the functional \. Thus if **a** is an array, **\+a** denotes the sum over the array. More generally $\backslash\Phi x$ for some dyadic operator Φ means $x_0\Phi(x_1\Phi..(x_n\Phi\iota))$ where ι is the implicit value given the operator and the type. Thus we can write \+ for $\sum$, * for $\prod$ etc. The dot product of two vectors can thus be written as

x:= \+ y*z;

instead of

x:=0;

for i:=0 to n do x:= x+ y[i]*z[i];

A reduction operation takes an argument of rank *r* and returns an argument of rank *r-1* except in the case where its argument is of rank 0, in which case it acts as the identity operation. Reduction is always performed along the last array dimension of its argument.

<operator reduction>	'\'<dyadic op> <multiplicative expression>

<dyadic op>	<expop> <multop> <addop>

The reserved word **rdu** is available as a lexical alternative to \, so \+ is equivalent to **rdu+**.

4.1.5 Complex conversion

Complex numbers can be produced from reals using the function **cmplx**. **cmplx(***re,im***)** is the complex number with real part *re*, and imaginaray part *im*.

The real and imaginary parts of a complex number can be obtained by the functions **re** and **im**. **re**(*c*) is the real part of the complex number *c*. **im**(*c*) is the imaginary part of the complex number *c*.

4.1.6 Conditional expressions

The conditional expression allows two different values to be returned depenent upon a boolean expression.

```
var a:array[0..63] of real;
...

a:=if a>0 then a else -a;

...
```

The **if** expression can be compiled in two ways:

1. Where the two arms of the if expression are parallelisable, the condition and both arms are evaluated and then merged under a boolean mask. Thus, the above assignment would be equivalent to:

   ```
   a:= (a and (a>0))or(not (a>0) and -a);
   ```

   were the above legal Pascal[2].

2. If the code is not paralleliseable it is translated as equivalent to a standard if statement. Thus, the previous example would be equivalent to:

   ```
   for i:=0 to 63 do if a[i]>0 then a[i]:=a[i] else a[i]:=-a[i];
   ```

   Expressions are non parallelisable if they include function calls.

The dual compilation strategy allows the same linguistic construct to be used in recursive function definitions and parallel data selection.

4.1.7 Factor

A factor is an expression that optionally performs exponentiation. Vector Pascal supports exponentiation either by integer exponents or by real exponents. A number x can be raised to an integral power y by using the construction x **pow** y. A number can be raised to an arbitrary real power by the ** operator. The result of ** is always real valued.

<expop>	'pow' '**'

<factor>	<unary expression> [<expop> <unary expression>]

4.1.8 Multiplicative expressions

Multiplicative expressions consist of factors linked by the multiplicative operators ***, /, div, mod, shr, shl and***. The use of these operators is summarised in table 4.1.

<multop>	'*' '/' 'div' 'shr' 'shl' 'and' 'mod'

<multiplicative expression>	<factor> [<multop> <factor>]* <factor>'in'<multiplicative expression>

[2]This compilation strategy requires that true is equivalent to -1 and false to 0. This is typically the representation of booleans returned by vector comparison instructions on SIMD instruction sets. In Vector Pascal this representation is used generally and in consequence, `true<false`.

Table 4.1: Multiplicative operators

Operator	Left	Right	Result	Effect of *a* **op** *b*
*	integer	integer	integer	multiply
	real	real	real	multiply
	complex	complex	complex	multiply
/	integer	integer	real	division
	real	real	real	division
	complex	complex	complex	division
div	integer	integer	integer	division
mod	integer	integer	integer	remainder
and	boolean	boolean	boolean	logical and
shr	integer	integer	integer	shift *a* by *b* bits right
shl	integer	integer	integer	shift *a* by *b* bits left
in	*t*	**set of** *t*	boolean	true if *a* is member of *b*

Table 4.2: Addition operations

	Left	Right	Result	Effect of *a* **op** *b*
+	integer	integer	integer	sum of *a* and *b*
	real	real	real	sum of *a* and *b*
	complex	complex	complex	sum of *a* and *b*
	set	set	set	union of *a* and *b*
	string	string	string	concatenate *a* with *b* 'ac'+'de'='acde'
-	integer	integer	integer	result of subtracting *b* from *a*
	real	real	real	result of subtracting *b* from *a*
	complex	complex	complex	result of subtracting *b* from *a*
	set	set	set	complement of *b* relative to *a*
+:	0..255	0..255	0..255	saturated + clipped to 0..255
	-128..127	-128..127	-128..127	saturated + clipped to -128..127
-:	0..255	0..255	0..255	saturated - clipped to 0..255
	-128..127	-128..127	-128..127	saturated - clipped to -128..127
min	integer	integer	integer	returns the lesser of the numbers
	real	real	real	returns the lesser of the numbers
max	integer	integer	integer	returns the greater of the numbers
	real	real	real	returns the greater of the numbers
or	boolean	boolean	boolean	logical or

Additive expressions

An additive expression allows multiplicative expressions to be combined using the addition operators **+, -, or, +:,max, min, -:**. The additive operations are summarised in table4.2 .

<addop>	'+' '-' 'or' 'max' 'min' '+:' '-:'

<additive expression>	<multiplicative expression> [<addop> <multiplicative expression>]*

Figure 4.1: Defining operations on complex numbers

```
type
complex = record data: array[0..1] of real;end;
var complexzero,complexone:complex;

{ headers for functions onto the complex numbers }
function cmplx(realpart,imag:real):complex;
function complex_add(A,B:Complex):complex;
function complex_conjugate(A:Complex):complex;
function complex_subtract(A,B:Complex):complex;
function complex_multiply(A,B:Complex):complex;
function complex_divide(A,B:Complex):complex;
function im(c:complex):real;
function re(c:complex):real;
{Standard operators on complex numbers }
{ symbol function identity element }
operator + = complex_add, complexzero;
operator / = complex_divide, complexone;
operator * = complex_multiply, complexone;
operator - = complex_subtract, complexzero;
```

Note that only the function headers are given here as this code comes from the interface part of the system unit. The function bodies and the initialisation of the variables complexone and complexzero are handled in the implementation part of the unit.

<expression>	<additive expression> <relational operator> <expression>

4.1.9 Operator overloading

The dyadic operators can be extended to operate on new types by operator overloading. Figure 4.1 shows how arithmetic on the type **complex** required by Extended Pascal [11] is defined in Vector Pascal. Each operator is associated with a semantic function and an identity element. The operator symbols must be drawn from the set of predefined Vector Pascal operators, and when expressions involving them are parsed, priorities are inherited from the predefined operators. The type signature of the operator is deduced from the type of the function[3]. When parsing expressions, the compiler first tries to resolve operations in terms of the predefined operators of the language, taking into account the standard mechanisms allowing operators to work on arrays. Only if these fail does it search for an overloaded operator whose type signature matches the context.

In the example in figure 4.1, complex numbers are defined to be records containing an array of reals, rather than simply as an array of reals. Had they been so defined, the operators `+`, `*`, `-`, `/` on reals would have masked the corresponding operators on complex numbers.

The provision of an identity element for complex addition and subtraction ensures that unary minus, as in $-x$ for x :complex, is well defined, and correspondingly that unary / denotes complex reciprocal. Overloaded operators can be used in array maps and array reductions.

[3]Vector Pascal allows function results to be of any non-procedural type.

4.2 Statements

<statement>	<variable>':='<expression> <procedure statement> <empty statement> 'goto' <label>; 'exit'['('<expression>')'] 'begin' <statement>[;<statement>]*'end' 'if'<expression>'then'<statement>['else'<statement>] <case statement> 'for' <variable>:= <expression> 'to' <expression> 'do' <statement> 'for' <variable>:= <expression> 'downto' <expression> 'do' <statement> 'repeat' <statement> 'until' <expression> 'with' <record variable> 'do' < statement> <io statement> 'while' <expression> 'do' <statement>

4.2.1 Assignment

An assignment replaces the current value of a variable by a new value specified by an expression. The assignment operator is :=. Standard Pascal allows assignment of whole arrays. Vector Pascal extends this to allow consistent use of mixed rank expressions on the right hand side of an assignment. Given

r0:real; r1:array[0..7] of real; r2:array[0..7,0..7] of real

then we can write

1. r**1:= r2[3]; { supported in standard Pascal }**
2. **r1:= /2; { assign 0.5 to each element of r1 }**
3. **r2:= r1*3; { assign 1.5 to every element of r2}**
4. **r1:= \+ r2; { r1gets the totals along the rows of r2}**
5. **r1:= r1+r2[1];{ r1 gets the corresponding elements of row 1 of r2 added to it}**

The assignment of arrays is a generalisation of what standard Pascal allows. Consider the first examples above, they are equivalent to:

1. **for i:=0 to 7 do r1[i]:=r2[3,i];**
2. **for i:=0 to 7 do r1[i]:=/2;**
3. **for i:=0 to 7 do for j:=0 to 7 do r2[i,j]:=r1[j]*3;**
4. **for i:=0 to 7 do begin t:=0; for j:=7 downto 0 do t:=r2[i,j]+t; r1[i]:=t; end;**
5. **for i:=0 to 7 do r1[i]:=r1[i]+r2[1,i];**

In other words the compiler has to generate an implicit loop over the elements of the array being assigned to and over the elements of the array acting as the data-source. In the above **i,j,t** are assumed to be temporary

variables not referred to anywhere else in the program. The loop variables are called implicit indices and may be accessed using **iota**.

The variable on the left hand side of an assignment defines an array context within which expressions on the right hand side are evaluated. Each array context has a rank given by the number of dimensions of the array on the left hand side. A scalar variable has rank 0. Variables occurring in expressions with an array context of rank *r* must have *r* or fewer dimensions. The *n* bounds of any *n* dimensional array variable, with $n \leq r$ occurring within an expression evaluated in an array context of rank *r* must match with the rightmost *n* bounds of the array on the left hand side of the assignment statement.

Where a variable is of lower rank than its array context, the variable is replicated to fill the array context. This is shown in examples 2 and 3 above. Because the rank of any assignment is constrained by the variable on the left hand side, no temporary arrays, other than machine registers, need be allocated to store the intermediate array results of expressions.

4.2.2 Procedure statement

A procedure statement executes a named procedure. A procedure statement may, in the case where the named procedure has formal parameters, contain a list of actual parameters. These are substituted in place of the formal parameters contained in the declaration. Parameters may be value parameters or variable parameters.

Semantically the effect of a value parameter is that a copy is taken of the actual parameter and this copy substituted into the body of the procedure. Value parameters may be structured values such as records and arrays. For scalar values, expressions may be passed as actual parameters. Array expressions are not currently allowed as actual parameters.

A variable parameter is passed by reference, and any alteration of the formal parameter induces a corresponding change in the actual parameter. Actual variable parameters must be variables.

<parameter>	<variable> <expression>	for formal parameters declared as var for other formal parameters

<procedure statement>	<identifier> <identifier> '(' <parameter> [','<parameter>]* ')'

Examples

1. **printlist;**
2. **compare(avec,bvec,result);**

4.2.3 Goto statement

A goto statement transfers control to a labelled statement. The destination label must be declared in a label declaration. It is illegal to jump into or out of a procedure.

Example goto 99;

4.2.4 Exit Statement

An exit statement transfers control to the calling point of the current procedure or function. If the exit statement is within a function then the exit statement can have a parameter: an expression whose value is returned from the function.

Examples

1. **exit;**
2. **exit(5);**

4.2.5 Compound statement

A list of statements separated by semicolons may be grouped into a compound statement by bracketing them with **begin** and **end** .

Example begin a:=x*3; b:=sqrt a end;

4.2.6 If statement

The basic control flow construct is the if statement. If the boolean expression between **if** and **then** is true then the statement following **then** is followed. If it is false and an else part is present, the statement following **else** is executed.

4.2.7 Case statement

The case statement specifies an expression which is evaluated and which must be of integral or ordinal type. Dependent upon the value of the expression control transfers to the statement labelled by the matching constant.

<case statement>	'case'<expression>'of'<case list>'end'

<case list>	<case list element>[';'<case list element.]*

<case list element>	<case label>[',' <case label>]':'<statement>

<case label>	<constant> <constant> '..' <constant>

Examples

```
case i of                case c of
1:s:=abs s;              'a':write('A');
2:s:= sqrt s;            'b','B':write('B');
3: s:=0                  'A','C'..'Z','c'..'z':write(' ');
end                      end
```

4.2.8 With statement

Within the component statement of the with statement the fields of the record variable can be referred to without prefixing them by the name of the record variable. The effect is to import the component statement into the scope defined by the record variable declaration so that the field-names appear as simple variable names.

Example **var s:record x,y:real end;**
begin
with s do begin x:=0;y:=1 end ;
end

4.2.9 For statement

A for statement executes its component statement repeatedly under the control of an iteration variable. The iteration variable must be of an integral or ordinal type. The variable is either set to count up through a range or down through a range.

for i:= e1 to e2 do s
is equivalent to
i:=e1; temp:=e2;while i<=temp do s;
whilst
for i:= e1 downto e2 do s
is equivalent to
i:=e1; temp:=e2;while i>= temp do s;

4.2.10 While statement

A while statement executes its component statement whilst its boolean expression is true. The statement

while e do s
is equivalent to
10: if not e then goto 99; s; goto 10; 99:

4.2.11 Repeat statement

A repeat statement executes its component statement at least once, and then continues to execute the component statement until its component expression becomes true.

repeat s until e
is equivalent to
10: s;if e then goto 99; goto 10;99:

4.3 Input Output

<io statement>	'writeln'[<outparamlist>] 'write'<outparamlist> 'readln'[<inparamlist>] 'read'<inparamlist>

<outparamlist>	'('<outparam>[','<outparam>]*')'

<outparam>	<expression>[':' <expression>] [':'<expression>]

<inparamlist>	'('<variable>[','<variable>]*')'

Input and output are supported from and to the console and also from and to files.

4.3.1 Input

The basic form of input is the **read** statement. This takes a list of parameters the first of which may optionally be a file variable. If this file variable is present it is the input file. In the absence of a leading file variable the input file is the standard input stream. The parameters take the form of variables into which appropriate translations of textual representations of values in the file are read. The statement

read(*a,b,c***)**

where *a,b,c* are non file parameters is exactly equivalent to the sequence of statements

read(*a***);read(***b***);read(***c***)**

The **readln** statement has the same effect as the read statement but finishes by reading a new line from the input file. The representation of the new line is operating system dependent. The statement

readln(*a,b,c***)**

where *a,b,c* are non file parameters is thus exactly equivalent to the sequence of statements

read(*a***);read(***b***);read(***c***);readln;**

Allowed typed for read statements are: integers, reals, strings and enumerated types.

4.3.2 Output

The basic form of output is the **write** statement. This takes a list of parameters the first of which may optionally be a file variable. If this file variable is present it is the output file. In the absence of a leading file variable the output file is the console. The parameters take the form of expressions whose values whose textual representations are written to the output file. The statement

write(*a,b,c***)**

where *a,b,c* are non file parameters is exactly equivalent to the sequence of statements

write(*a***);write(***b***);write(***c***)**

The **writeln** statement has the same effect as the write statement but finishes by writing a new line to the output file. The representation of the new line is operating system dependent. The statement

writeln(*a,b,c***)**

where *a,b,c* are non file parameters is thus exactly equivalent to the sequence of statements

write(*a***);write(***b***);write(***c***);writeln;**

Allowed types for write statements are integers, reals, strings and enumerated types.

Parameter formating

A non file parameter can be followed by up to two integer expressions prefixed by colons which specify the field widths to be used in the output. The write parameters can thus have the following forms:

e e:*m e*:*m*:*n*

1. If *e* is an integral type its decimal expansion will be written preceeded by sufficient blanks to ensure that the total textual field width produced is not less than *m*.
2. If *e* is a real its decimal expansion will be written preceeded by sufficient blanks to ensure that the total textual field width produced is not less than *m*. If *n* is present the total number of digits after the decimal point will be *n*. If *n* is omitted then the number will be written out in exponent and mantissa form with 6 digits after the decimal point
3. If *e* is boolean the strings 'true' or 'false' will be written into a field of width not less than m.
4. If *e* is a string then the string will be written into a field of width not less than *m*.

Chapter 5

Programs and Units

Vector Pascal supports the popular system of separate compilation units found in Turbo Pascal. A compilation unit can be either a program, a unit or a library.

<program>	'program' <identifier>';'[<uses>';']<block>'.'

<uses>	'uses' <identifier>[','<identifier>]

<block>	[<decls>';']*'begin' <statement>[';'<statement>]*'end'

<decls>	'const' <constant declaration>[';'<constant declaration>]* 'type'<type definition>[';'<type definition>]* 'label' <label>[',' <label>] <procedure declaration> 'var' <variable declaration>[';' <variable declaration>]

<unit>	<unit type> <identifier>'interface' <decls><uses> 'implementation'<block>'.'

<unit type>	'unit' 'library'

An executable compilation unit must be declared as a program. The program can use several other compilation units all of which must be either units or libraries. The units or libraries that it directly uses are specified by a list of identifiers in an optional use list at the start of the program. A unit or library has two declaration portions and an executable block.

5.1 The export of identifiers from units

The first declaration portion is the interface part and is preceded by the reserved word **interface**.

The definitions in the interface section of unit files constitute a sequence of enclosing scopes, such that successive units in the with list ever more closely contain the program itself. Thus when resolving an identifier,

if the identifier can not be resolved within the program scope, the declaration of the identifier within the interface section of the rightmost unit in the uses list is taken as the defining occurrence. It follows that rightmost occurrence of an identifier definition within the interface parts of units on the uses list overrides all occurrences in interface parts of units to its left in the uses list.

The implementation part of a unit consists of declarations, preceded by the reserved word **implementatio**n that are private to the unit with the exception that a function or procedure declared in an interface context can omit the procedure body, provided that the function or procedure is redeclared in the implementation part of the unit. In that case the function or procedure heading given in the interface part is taken to refer to the function or procedure of the same name whose body is declared in the implementation part. The function or procedure headings sharing the same name in the interface and implementation parts must correspond with respect to parameter types, parameter order and, in the case of functions, with respect to return types.

A unit may itself contain a use list, which is treated in the same way as the use lists of a program. That is to say, the use list of a unit makes accessible identifiers declared within the interface parts of the units named within the use list to the unit itself.

5.1.1 The export of procedures from libraries.

If a compilation unit is prefixed by the reserved word **library** rather than the words **program** or **unit**, then the procedure and function declarations in its interface part are made accessible to routines written in other languages.

5.1.2 The export of Operators from units

A unit can declare a type and export operators for that type.

5.2 The invocation of programs and units

Programs and units contain an executable block. The rules for the execution of these are as follows:

1. When a program is invoked by the operating system, the units or libraries in its use list are invoked first followed by the executable block of the program itself.

2. When a unit or library is invoked, the units or libraries in its use list are invoked first followed by the executable block of the unit or library itself.

3. The order of invocation of the units or libraries in a use list is left to right with the exception provided by rule 4.

4. No unit or library may be invoked more than once.

Note that rule 4 implies that a unit x to the right of a unit y within a use list, may be invoked before the unit y, if the unit y or some other unit to y's left names x in its use list.

Note that the executable part of a library will only be invoked if the library in the context of a Vector Pascal program. If the library is linked to a main program in some other language, then the library and any units that it uses will not be invoked. Care should thus be taken to ensure that Vector Pascal libraries to be called from main programs written in other languages do not depend upon initialisation code contained within the executable blocks of units.

5.3 The compilation of programs and units.

When the compiler processes the use list of a unit or a program then, from left to right, for each identifier in the use list it attempts to find an already compiled unit whose filename prefix is equal to the identifier. If such a file exists, it then looks for a source file whose filename prefix is equal to the identifier, and whose suffix is `.pas`. If such a file exists and is older than the already compiled file, the already compiled unit, the compiler loads the definitions contained in the pre-compiled unit. If such a file exists and is newer than the pre-compiled unit, then the compiler attempts to re-compile the unit source file. If this recompilation proceeds without the detection of any errors the compiler loads the definitions of the newly compiled unit. The definitions in a unit are saved to a file with the suffix `.mpu`, and prefix given by the unit name. The compiler also generates an assembler file for each unit compiled.

5.3.1 Linking to external libraries

It is possible to specify to which external libraries - that is to say libraries written in another languge, a program should be linked by placing in the main program linkage directives. For example

{$linklib ncurses}

would cause the program to be linked to the ncurses library.

5.4 The System Unit

All programs and units include by default the unit system.pas as an implicit member of their with list. This contains declarations of private run time routines needed by Vector Pascal and also the following user accessible routines.

procedure **append(var f:file);** This opens a file in append mode.

procedure **assign(var f:file;var fname:string);** Associates a file name with a file. It does not open the file.

procedure **blockread(var f:file;var buf;count:integer; var resultcount:integer);** Trys to read count bytes from the file into the buffer. Resultcount contains the number actually read.

LatexCommand \index{blockwrite}procedure blockwrite(var f:file;var buf;count:integer; var resultcount:integer); Write count bytes from the buffer. Resultcount gives the number actually read.

procedure **close**(var f:file); Closes a file.

function **eof**(var f:file):boolean; True if we are at the end of file f.

procedure **erase**(var f:file); Delete file f.

function **eoln**(var f:file):boolean; True if at the end of a line.

function **exp**(d:real):real; Return e^x

function **filesize**(var f: fileptr):integer; Return number of bytes in a file.

function **filepos**(var f:fileptr):integer; Return current position in a file.

procedure **freemem(var p:pointer; num:integer);** Free num bytes of heap store. Called by dispose.

bold procedure getmem(var p:pointer; num:integer); Allocate num bytes of heap. Called by new.

procedure **gettime(var hour,min,sec,hundredth:integer);** Return time of day.

Return the integer part of r as a real.

function **ioresult:integer;** Returns a code indicating if the previous file operation completed ok. Zero if no error occurred.

function **length(var s:string):integer;** Returns the length of s.

procedure **pascalexit(code:integer);** Terminate the program with code.

Time in 1/100 seconds since program started.

function **random:integer;** Returns a random integer.

procedure **randomize;** Assign a new time dependent seed to the random number generator.

procedure **reset(var f:file);** Open a file for reading.

procedure **rewrite(var f :file);** Open a file for writing.

function **trunc(r:real):integer;** Truncates a real to an integer.

Chapter 6

Implementation issues

The compiler is implemented in java to ease portability between operating systems.

6.1 Invoking the compiler

The compiler is invoked with the command

```
vpc filename
```

where filename is the name of a Pascal program or unit. For example

```
vpc test
```

will compile the program test.pas and generate an executable file `test`, (`test.exe` under windows).

The command `vpc` is a shell script which invokes the java runtime system to execute a `.jar` file containing the compiler classes. Instead of running vpc the java interpreter can be directly invoked as follows

```
java -jar mmpc.jar filename
```

The `vpc` script sets various compiler options appropriate to the operating system being used.

6.1.1 Environment variable

The environment variable `mmpcdir` must be set to the directory which contains the `mmpc.jar` file, the runtime library `rtl.o` and the `system.pas` file.

6.1.2 Compiler options

The following flags can be supplied to the compiler :

`-Afilename` Defines the assembler file to be created. In the absence of this option the assembler file is `p.asm`.

`-Ddirname` Defines the directory in which to find `rtl.o` and `system.pas`.

`-V` Causes the code generator to produce a verbose diagnostic listing to `foo.lst` when compiling `foo.pas`.

Table 6.1: Code generators supported

CGFLAG	description
IA32	generates code for the Intel 486 instruction-set
Pentium	generates code for the Intel P6 with MMX instruction-set
K6	generates code for the AMD K6 instruction-set, use for Athlon
P3	generates code for the Intel PIII processor family
P4	generates code for the Intel PIV family and Athlon XP

`-oexefile` Causes the linker to output to `exefile` instead of the default output of `p.exe`.

`-U` Defines whether references to external procedures in the assembler file should be preceded by an under-bar '_'. This is required for the coff object format but not for elf.

`-S` Suppresses assembly and linking of the program. An assembler file is still generated.

`-fFORMAT` Specifies the object format to be generated by the assembler. The object formats currently used are elf when compiling under Unix or when compiling under windows using the cygwin version of the gcc linker, or coff when using the djgpp version of the gcc linker. for other formats consult the NASM documentation.

`-cpuCGFLAG` Specifies the code generator to be used. Currently the code generators shown in table 6.1 are supported.

6.1.3 Dependencies

The Vector Pascal compiler depends upon a number of other utilities which are usually pre-installed on Linux systems, and are freely available for Windows systems.

NASM The net-wide assembler. This is used to convert the output of the code generator to linkable modules. It is freely available on the web for Windows.

gcc The GNU C Compiler, used to compile the run time library and to link modules produced by the assembler to the run time library.

java The java virtual machine must be available to interpret the compiler. The a number of java interpreters and just in time compilers are freely available for Windows.

6.2 Compiler Structure

The main program class of the compiler ilcg.Pascal.PascalCompiler translates the source code of the program into an internal structure called an ILCG tree [7]. A machine generated code generator then translates this into assembler code. An example would be the class ilcg.tree.IA32. An

assembler and linker specified in descendent class of the code generator then translate the assembler code into an executable file.

To port the compiler to a new machine, say a P5, it is necessary to

1. Write a new machine description `P5.ilc` in ILCG source code.
2. Compile this to a code generator in java with the ilcg compiler generator using a command of the form

   (a) `java ilcg.ILCG cpus/P5.ilc ilcg/tree/P5.java P5`
3. Write an interface class ilcg/tree/P5CG which is a subclass of P5 and which invokes the assembler and linker.

Sample machine descriptions and interface classes are available on request to those wishing to port the compiler.

6.3 Calling conventions

Procedure parameters are passed using a modified C calling convention to facilitate calls to external C procedures. Parameters are pushed on to the stack from right to left. Value parameters are pushed entire onto the stack, var parameters are pushed as addresses.

Example

```
unit callconv;
interface
type intarr= array[1..8] of integer;
procedure foo(var a:intarr; b:intarr; c:integer);
implementation
procedure foo(var a:intarr; b:intarr; c:integer);
begin
end;
var x,y:intarr;
begin
      foo(x,y,3);
end.
```

This would generate the following code for the procedure foo.

```
; procedure generated by code genera-
tor class ilcg.tree.PentiumCG
le8e68de10c5:
;        foo
 enter   spaceforfoo-4*1,1
;8
 le8e68de118a:
spaceforfoo equ 4
;.... code for foo goes here
fooexit:
leave
 ret 0
```

and the calling code is

```
push DWORD      3          ; push rightmost argument
lea esp,[  esp-32]         ; create space for the array
mov DWORD  [  ebp -52],0   ; for loop to copy the array
le8e68de87fd:              ; the loop is
                           ; unrolled twice and
cmp DWORD  [  ebp-52], 7   ; parallelised to copy
                           ; 16 bytes per cycle
jg near  le8e68de87fe
mov ebx,DWORD  [  ebp  -52]
imul   ebx,    4
movq MM1, [  ebx+ le8e68dddaa2-48]
movq  [  esp+ebx],MM1
mov eax,DWORD  [  ebp+     -52]
lea ebx,[  eax+     2]
imul   ebx,    4
movq MM1, [  ebx+ le8e68dddaa2 -48]
movq  [  esp+ebx],MM1
lea ebx,[  ebp+     -52]
add  DWORD  [ ebx],     4
jmp  le8e68de87fd
le8e68de87fe:               ; end of array
                            ; copying loop
push DWORD  le8e68dddaa2-32 ; push the address of the
                            ; var parameter
EMMS                        ; clear MMX state
 call le8e68de10c5          ; call the local
                            ; label for foo
add esp, 40                 ; free space on the stack
```

Function results

Function results are returned in registers for scalars following the C calling convention for the operating system on which the compiler is implemented. Records, strings and sets are returned by the caller passing an implicit parameter containing the address of a temporary buffer in the calling environment into which the result can be assigned. Given the following program

```
program
type t1= set of char;
var x,y:t1;
function bar:t1;begin bar:=y;end;

begin
     x:=bar;
end.
```

The call of bar would generate

```
push ebp
add  dword[esp] , -128     ; address of buffer on stack
call le8eb6156ca8          ; call bar to place
                           ; result in buffer
add esp, 4                 ; discard the address
mov DWORD  [  ebp+ -132], 0; for loop to copy
                           ; the set 16 bytes
le8eb615d99f:              ; at a time into x using the
                           ; MMX registers
cmp DWORD  [  ebp+     -132],  31
jg near  le8eb615d9910
mov ebx,DWORD  [  ebp+     -132]
movq MM1, [  ebx+ebp +     -128]
movq  [  ebx+ebp +     -64],MM1
mov eax,DWORD  [  ebp+     -132]
lea ebx,[  eax+     8]
```

```
movq MM1, [  ebx+ebp +     -128]
movq  [  ebx+ebp +     -64],MM1
lea ebx,[  ebp+     -132]
add  DWORD  [ ebx],     16
jmp  le8eb615d99f
le8eb615d9910:
```

6.4 Array representation

A static array is represented simply by the number of bytes required to store the array being allocated in the global segment or on the stack.

A dynamic array is always represented on the heap. Since its rank is known to the compiler what needs to be stored at run time are the bounds and the means to access it. For simplicity we make the format of dynamic and conformant arrays the same. Thus for schema

s(a,b,c,d:integer)= array[a..b,c..d] of integer

whose run time bounds are evaluated to be 2..4,3..7 we would have the following structure:

address	field	value
x	base of data	address of first integer in the array
x+4	a	2
x+8	b	4
x+12	step	20
x+16	c	3
x+20	d	7

The base address for a schematic array on the heap, will point at the first byte after the array header show. For a conformant array, it will point at the first data byte of the array or array range being passed as a parameter. The step field specifies the length of an element of the second dimension in bytes. It is included to allow for the case where we have a conformant array formal parameter

x:array[a..b:integer,c..d:integer] of integer

to which we pass as actual parameter the range

p[2..4,3..7]

as actual parameter, where **p:array[1..10,1..10] of integer**

In this case the base address would point at @p[2,3] and the step would be 40 - the length of 10 integers.

6.4.1 Range checking

When arrays are indexed, the compiler plants run time checks to see if the indices are within bounds. In many cases the optimiser is able to remove these checks, but in those cases where it is unable to do so, some performance degradation can occur. Range checks can be disabled or enabled by the compiler directives.

{$r-} { disable range checks }

{$r+} { enable range checks }

Performance can be further enhanced by the practice of declaring arrays to have lower bounds of zero. The optimiser is generally able to generate more efficient code for zero based arrays.

Index

Bibliography

[1] Advanced Micro Devices, 3DNow! Technology Manual, 1999.

[2] Aho, A.V., Ganapathi, M, TJiang S.W.K., Code Generation Using Tree Matching and Dynamic Programming, ACM Trans, Programming Languages and Systems 11, no.4, 1989, pp.491-516.

[3] Burke, Chris, J User Manual, ISI, 1995.

[4] Cattell R. G. G., Automatic derivation of code generators from machine descriptions, ACM Transactions on Programming Languages and Systems, 2(2), pp. 173-190, April 1980.

[5] Chaitin. G., Elegant Lisp Programs, in The Limits of Mathematics, pp. 29-56, Springer, 1997.

[6] Cheong, G., and Lam, M., An Optimizer for Multimedia Instruction Sets, 2nd SUIF Workshop, Stanford University, August 1997.

[7] Cockshott, Paul, Direct Compilation of High Level Languages for Multi-media Instruction-sets, Department of Computer Science, University of Glasgow, Nov 2000.

[8] Susan L. Graham, Table Driven Code Generation, IEEE Computer, Vol 13, No. 8, August 1980, pp 25..37.

[9] Intel, Intel Architecture Software Developers Manual Volumes 1 and 2, 1999.

[10] Intel, Willamette Processor Software Developer's Guide, February, 2000.

[11] ISO, Extended Pascal ISO 10206:1990, 1991.

[12] Jensen, K., Wirth, N., PASCAL User Manual and Report, Springer 1978.

[13] K. E. Iverson, A Programming Language, John Wiley & Sons, Inc., New York (1962), p. 16.

[14] Iverson, K. E. . Notation as a tool of thought. Communications of the ACM, 23(8), 444-465, 1980.

[15] Iverson K. E, A personal View of APL, IBM Systems Journal, Vol 30, No 4, 1991.

[16] Iverson, Kenneth E., J Introduction and Dictionary, Iverson Software Inc. (ISI), Toronto, Ontario, 1995. 4, pp 347-361, 2000.

[17] Leupers, R., Compiler Optimization for Media Processors, EMMSEC 99/Sweden 1999.

[18] Metcalf, M., and Reid., J., The F Programming Language, OUP, 1996.

[19] Peleg, A., Wilke S., Weiser U., Intel MMX for Multimedia PCs, Comm. ACM, vol 40, no. 1 1997.

[20] Srereman, N., and Govindarajan, G., A Vectorizing Compiler for Multimedia Extensions, International Journal of Parallel Programming, Vol. 28, No. 4, pp 363-400, 2000.

[21] Étienne Gagnon, SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK, School of Computer Science McGill University, Montreal, March 1998.

[22] Texas Instruments, TMS320C62xx CPU and Instruction Set Reference Guide, 1998.