Vector Pascal, ILCG and Viper

Paul Cockshott and Ken Renfrew

13th December 2002

Contents

Introduction

Vector Pascal is a dialect of Pascal designed as part of a research project that aims to make efficient use of the multi-media instructionsets of recent processors. It supports data parallel operations and saturated arithmetic. It was developed at the Image and Graphics Lab at the Department of Computing Science at the University of Glasgow. This lab is part of the Imaging Faraday Partnership and funded by the Department of Trade and Industry, having as its objective the transfer of research technology in the field of computer imaging to UK industry. The language was developed because in the late 90s we found that there was inadequate compiler support for the MMX instruction set which we wanted to use for our stereo vision software.

The project to develop the Vector Pascal compiler has involved two components: on the one hand we have developed a language (ILCG) that can be used to describe the semantics of SIMD machines. Associated with this we developed a code-generator generator that takes ILCG descriptions of a machine and writes an optimising code generator for that machine in Java. Secondly we set about developing a Pascal compiler to act as a testbed for this code-generator technology. It soon became evident that to make best use of data-parallelism a number of extensions to ISO Pascal would be desirable, analogous to the extensions made to Fortran when it moved to support data parallel operations in Fortran 90. This manual describes the Vector Pascal language along with the ILCG language used to port it to new processors.

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[35] and in general purpose CPU chips like the Intel IA32[14] or the AMD K6 [2].

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[13][29] 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[8][26][24][32]. 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[17].

Define a vector of type T as having type $T[]$. Then if we have a binary operator X:(T , T)$\rightarrow T$, in languages derived from APL we automatically have an operator X:(T[] ,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[19][5][20], and F[28] 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[7]. 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[18] 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[21]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.

Part I

Language Reference Manual

Paul Cockshott

Chapter 1

Elements of the language

1.1 Alphabet

The Vector Pascal compiler accepts files in the UTF-8 encoding of Unicode as source. Since ASCII is a subset of this, ASCII files are valid input.

Vector Pascal programs are made up of letter, digits and special symbols. The letters digits and special symbols are draw either from a base character set or from an extended character set. The base character set is drawn from ASCII and restricts the letters to be from the Latin alphabet. The extended character set allows letters from other alphabets.

The special symbols used in the base alphabet are shown in table1.1 .

1.1.1 Extended alphabet

The extended alphabet is described in Using Unicode with Vector Pascal.

1.2 Reserved words

The reserved words are

```
ABS, ADDR, AND, ARRAY,
BEGIN, BYTE2PIXEL,
CASE, CAST, CDECL, CHR, CONST, COS,
DISPOSE, DIV, DO, DOWNTO,
END, ELSE, EXIT, EXTERNAL,
```

Table 1.1: Special symbols

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

```
   FALSE, FILE, FOR, FUNCTION,
   GOTO,
   IF, IMPLEMENTATION, IN, INTERFACE, IOTA,
   LABEL, LIBRARY, LN,
   MAX, MIN, MOD,
   NAME, NDX, NEW, NOT,
   OF, OR, ORD, OTHERWISE,
   PACKED, PERM, PIXEL2BYTE, POW, PRED,
PROCEDURE, PROGRAM, PROTECTED ,
   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
```

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

1.3 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].

1.4 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. ISO Pascal allows the Latin letters A-Z to be used in identifiers. Vector Pascal extends this by allowing symbols from the Greek, Cyrillic, Katakana and Hiragana, or CJK character sets

1.5 Literals

1.5.1 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 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].

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

[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.

Table 1.2: 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

.

<digit sequence>	<digit> +

<decimal integer>	<digit sequence>

<hex integer>	'$'<hexdigit>+

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

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

1.5.2 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>	'-' '+'

<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.

1.5.3 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 any of the Unicode characters above u0020. The character strings must be input to the compiler in UTF-8 format.

Chapter 2

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.

2.1 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 table2.1 . This restricts simple constants to be of scalar or string types.

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

<typed constant>	<expression> <array constant>

Table 2.1: The operators permitted in Vector Pascal constant expressions.

+	-	*	/	div	mod	shr	shl	and	or

2.1.1 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>]* ')'

2.1.2 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.

2.2 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;
```

Table 2.2: 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

2.3 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>

2.3.1 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.

`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.

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

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

*	/

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

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

.

2.3.2 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' 'string(' <ingeger constant>')'

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 an ordered type, that is a type on which the operations $<$, $=$ and $>$ are defined[2]. Thus sets may be declared whose base types are characters, numbers, ordinals, or strings. Any user defined type on which the comparison operators have been defined can also be the base type of a set.

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

2.3.3 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[3]. 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

[2]ISO Pascal requires the base type to be a scalar type, a character type, integer type or a subrange thereof. When the base type is one of these, Vector Pascal implements the set using bitmaps. When the type is other than these, balanced binary trees are used. It is strongly recomended that use be made of Boehm garbage collector (see section 5.1.2) if non-bitmapped sets are used in a program.

[3]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.

be declared to point at dynamic arrays. A dynamic array is an array whose bounds are determined at run time.

Pascal 90[15] 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.

2.4 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.

Text files are assumed to be in Unicode UTF-8 format. Conversions are performed between the internal representation of characters and UTF-8 on input/output from/to a text file.

2.5 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>

Examples

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

2.5.1 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';
```

2.5.2 Entire Variables

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

2.5.3 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 ≤`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`.

2.5.4 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]);
```

2.5.5 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 3.1.3,3.2.1.

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
```

2.5.6 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>

2.5.7 Referenced Variables

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

<referenced variable>	<variable> '^'

2.6 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[4] 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;
```

Protected parameters

A parameter declaration may be prefixed by the word `PROTECTED`. A protected parameter ISO-10206
may not be assigned to within the body of the function. Protected parameters are useful for obtaining the semantic effect of a value parameter where efficiency considerations lead an array to be passed as a var parameter.

Parameter types

Standard Pascal requires the types of parameters to be given by type names. Where arrays are passed as parameters they must be of user defined array types. Vector Pascal allows array types to be explicitly given in the parameter declarations.

[4] see section 4.

Chapter 3

Algorithms

3.1 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.

3.1.1 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.

3.1.2 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.

3.1.3 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	operators	implicit value
number	+,-	0
string	+	''
set	+	empty 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

Table 3.1: Unary operators

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', '$\surd$'	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', 'ι'	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.

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])`

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

`succ` and `pred`

There is a pair of built in operators `succ` and `pred` defined over every scalar type t such that $x = \texttt{succ pred} x \forall x \in t$ and $y = \texttt{pred succ} y \forall y \in t$. VECTOR

This definition of the successor and predecessor functions differs from that given in the

```
PROGRAM scalars;
TYPE day=( sunday,   monday, tuesday, wednesday,
             thursday, friday, saturday);
VAR today,tomorrow,day2:day;
BEGIN
  today:= friday MAX saturday;
  tomorrow := SUCC today;
   day2:=SUCC(tommorrow,2);
  write(today, tomorrow,today < friday, today=saturday);
END.
```

output generated:

```
    saturday      sunday       false       tuesday
```

Figure 3.1: Which presents a program illustrating both the comparability of user defined scalar types and their cyclical nature.

Pascal Standard which defines `succ` as follows:

ISO-7185

> $succ(x)$
>
> The function shall yield a value whose ordinal number is one greater than that of the expression x, if such a value exists. It shall be an error if such a value does not exist. [1]

The implication of these definitions are that in Vector Pascal the successor function operates in a modulo fashion. As one steps through an scalar type with the successor function one eventually gets back to the starting point. This is illustrated in figure 3.1.

This modular operation of `succ` and `pred` can be switched off by using the {$M-} inline directive.

Vector Pascal follows ISO Extended Pascal in allowing a second integer parameter to the functions as illustrated in figure 3.1.

`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[2].

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;
```

[1] See [16] page 45.
[2] See section 3.2.1.

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`.

`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 3.1.3, the program fragment:

```
m1:= (trans v1)*v2;
writeln('(trans v1)*v2');
writeln(m1);
m2 := trans m1;
writeln('transpose 1..3,0..4 matrix');
writeln(m2);
```

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
```

3.1.4 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 summation, `\*` for nary product etc. The dot product of two vectors can thus be written as

```
x:= \+ y*x;
```

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.

The operations of summation and product can be be written eithter as the two functional forms $\backslash +$ and $\backslash *$ or as the prefix operators $\sum$ (Unicode 2211) and $\prod$ (Unicode 220f).

<operator reduction>	'\\'<dyadic op> <multiplicative expression> '$\sum$' <mutliplicative expression> '$\prod$' < multiplicative expression>

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

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

3.1.5 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.

3.1.6 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[3].

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.

Use of boolean mask vectors

In array programming many operations can be efficiently be expressed in terms of boolean mask vectors. Given the declarations:

```
const i:array[1..4] of integer=(2,4,6,8);
      r:array[1..4] of real =(1.0,1.1,1.2,1.4);
      b:array[1..4] of boolean=(true,false,true,false);
      s:array[1..4] of string=('from','the','masters','of');
```

[3] 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 3.2: Multiplicative operators

Operator	Left	Right	Result	Effect of *a* `op` *b*
`*`, ×	integer	integer	integer	multiply
	real	real	real	multiply
	complex	complex	complex	multiply
	string	integer	string	replication: 'ab'*2 = 'abab'
`/`	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*

then the boolean array `b` can be used to mask the other arrays such that

```
write(i and b);
write(r and b);
write(s and b);
```

produces

```
    2          0          6          0
    1          0        1.2          0
  from                masters
```

Anding a number or string with true leaves the number or string unchanged. Anding a number of string with false returns the additive identity element : 0 or the null string respectively.

3.1.7 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>]

3.1.8 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 3.2.

Table 3.3: 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`, $\wedge$	boolean	boolean	boolean	logical or
`><`	set	set	set	symetric difference

<multop>	'*' '$\times$' '/' 'div' '$\div$' 'shr' 'shl' 'and' '$\vee$' 'mod'

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

3.1.9 Additive expressions

An additive expression allows multiplicative expressions to be combined using the addition operators `+, -, or, +:,max, min, -:`, `><`. The additive operations are summarised in table3.3 .

<addop>	'+' '-' 'or' '$\wedge$' 'max' 'min' '+:' '-:'

Table 3.4: Relational operators

<	Less than
>	Greater than
<= , $\leq$	Less than or equal to
>= , $\geq$	Greater than or equal to
<>	Not equal to
=	Equal to

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

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

3.1.10 Expressions

An expression can optionally involve the use of a relational operator to compare the results of two additive expressions. Relational operators always return boolean results and are listed in table 3.4.

3.1.11 Operator overloading

The dyadic operators can be extended to operate on new types by operator overloading. Figure 3.2 shows how arithmetic on the type `complex` required by Extended Pascal [15] is defined in Vector Pascal. Each operator is associated with a semantic function and if it is a non-relational operator, 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[4].

<operator-declaration>	'operator' 'cast' '=' <identifier> 'operator' <dyadicop> '=' <identifier>','<identifier> 'operator' <relational operator> '=' <identifier>

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 3.2, 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.

Implicit casts

The Vector Pascal language already contains a number of implicit type conversions that are context determind. An example is the promotion of integers to reals in the context of

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

```
interface
  type
    Complex = record data :  array [0..1] of real ;
    end ;

  var
     complexzero, complexone : complex;

  function real2cmplx ( realpart :real ):complex ;
  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 ;
    { 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 ;
    operator cast = real2cmplx ;
```

Figure 3.2: Defining operations on complex numbers

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.

arithmetic expressions. The set of implicit casts can be added to by declaring an operator to be a cast as is shown in the line:

```
operator cast = real2cmplx ;
```

Given an implicit cast from type $t_0 \rightarrow t_1$, the function associated with the implicit cast is then called on the result of any expression $e : t_0$ whose expression context requires it to be of type t_1.

3.2 Statements

<statement>	<variable>':='<expression>
	<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>
	'for' <variable> 'in' <expression> 'do' <statement>
	'for' <variable> '∈' <expression> 'do' <statement>
	'repeat' <statement> 'until' <expression>
	'with' <record variable> 'do' < statement>
	<io statement>
	'while' <expression> 'do' <statement>

3.2.1 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. `r1:= 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; { r1 gets 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.

Unicode form

The Unicode ← can be used as an alternative to the ASCII sequence := for assignment.

3.2.2 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);`

3.2.3 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;`

3.2.4 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);`

3.2.5 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;`

3.2.6 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.

3.2.7 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 actions>'end'

<case actions>	<case list> <case list> 'else' <statement> <case list> 'otherwise' <statement>

<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
```

3.2.8 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
```

3.2.9 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;
```

for $\in$ do

A special form of for statement allows iteration through a set.

```
for x in y do s
```

or alternatively

```
for x ∈ y do s
```

will execute statement `s` for each element `x` of the set `y` in turn. The elements are selected in the order specified by the $<$ operator on the base type of the set. The varible `x` must conform to the base type of the set `y`.

3.2.10 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:
```

3.2.11 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:
```

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 Generic array output

In standard Pascal, any type including an array can be written to a binary file, but output to text files must be of simple types or string types. Vector Pascal, in conformity with the general overloading of operations on arrays, extends `write` to operate on arrays. A one dimension array is written as a sequence of values on a line. Thus for a one dimensional array a with bounds $l..u$ the statement `write(`a`);` would be equivalent to

`FOR` i`:=`l `TO` u `DO write(`a`[`i`]);`

Where i is a temporay integer variable. If the array *b* has more than one dimension, and the bounds of the leftmost dimension are $m..n$ then the expansion is:

`FOR` j`:=`m `TO` n `DO writeln(`b`[`j`]);`

In conjunction with the re-write rules given in above this gives an unambiguous definition of `write` for arrays. The effect is that dimensions of the arrays are separated by newlines.

Array input is also supported with the corresponding read statements.

3.3.4 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 4

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>’.’

<invocation>	<identifier>[’(’ <type identifier>[’,’<type identifier>]*’)’]

<uses>	’uses’ <invocation>[’,’<invocation>]*

<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 header> <unit body>

<unit body>	’interface’[<uses>][<decls>] ’implementation’<block>’.’ ’interface’[<uses>][<decls>] ’in’ <invocation> ’;’

<unit header>	<unit type><identifier> ’unit’ <identifier> ’(’ <type identifier> [’,’ <type identifier>]* ’)’

<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.

4.1 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

unit *genericsort(t)* ;
interface
type
    dataarray (*n* ,*m* :*integer*)=**array** [*n* ..*m*] **of** *t* ;
procedure *sort* (**var** *a* :*dataarray*); (see Figure 4.2)

implementation

procedure *sort* (**var** *a* :*dataarray*); (see Figure 4.2)
begin
end .

Figure 4.1: A polymorphic sorting unit.

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 `implementation` 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.

4.1.1 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.

4.1.2 The export of Operators from units

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

4.2 Unit parameterisation and generic functions

Standard Pascal provides es some limited support for polymorphism in its `read` and `write` functions. Vector Pascal allows the writing of polymorphic functions and procedures through the use of parameteric units.

A unit header can include an optional parameter list. The parameters identifiers which are interepreted as type names. These can be used to declare polymorphic procedures and functions, parameterised by these type names. This is shown in figure 4.1.

procedure *sort* (**var** *a* :*dataarray*);
var
Let *i*, *j* $\in$ integer;
Let *temp* $\in$ t;
begin
for $i \leftarrow a.n$ **to** $a.m - 1$ **do**
for $j \leftarrow a.n$ **to** $a.m - 1$ **do**
if $a_j > a_{j+1}$ **then** *begin* **begin**
$temp \leftarrow a_j$;
$a_j \leftarrow a_{j+1}$;
$a_{j+1} \leftarrow temp$;
end ;
end ;

Figure 4.2: procedure sort

4.3 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.

4.4 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.

4.4.1 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.

4.5 Instantiation of parametric units

Instantiation of a parametric unit refers to the process by which the unbound type variables introduced in the parameter list of the unit are bound to actual types. In Vector Pascal all instantiation of parametric units and all type polymorphism are resolved at compile time. Two mechanisms are provided by which a parametric unit may be instantiated.

4.5.1 Direct instantiation

If a generic unit is invoked in the use list of a program or unit, then the unit name must be followed by a list of type identifiers. Thus given the generic sort unit in figure 4.1, one could instantiate it to sort arrays of reals by writing

uses *genericsort*(*real*);

at the head of a program. Following this header, the procedure *sort* would be declared as operating on arrays of reals.

4.5.2 Indirect instantiation

A named unit file can indirectly instantiate a generic unit where its unit body uses the syntax

'interface' <uses><decls> 'in' <invocation> ';'

For example

unit *intsort* ;
interface
in *genericsort* (*integer*);

would create a named unit to sort integers. The naming of the parametric units allows more than one instance of a given parametric unit to be used in a program. The generic sort unit could be used to provide both integer and real sorting procedures. The different variants of the procedures would be distinquished by using fully qualified names - e.g., *intsort.sort*.

4.6 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 5

Implementation issues

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

5.1 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.

5.1.1 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.

5.1.2 Compiler options

The following flags can be supplied to the compiler :

`-L` Causes a latex listing to be produced of all files compiled. The level of detail can be controled using the codes -L1 to -L3, otherwise the maximum detail level is used.

`-OPT`*n* Sets the optimisation level attempted. -OPT0 is no optimisation, -OPT3 is the maximum level attempted. The default is -OPT1.

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

Table 5.1: Code generators supported

CGFLAG	description
IA32	generates code for the Intel 486 instruction-set uses the NASM assembler
Pentium	generates code for the Intel P6 with MMX instruction-set uses the NASM assembler
gnuPentium	generates code for the Intel P6 with MMX instruction-set using the `as` assembler in the gcc package
K6	generates code for the AMD K6 instruction-set, use for Athlon uses the NASM assembler
P3	generates code for the Intel PIII processor family uses the NASM assembler
P4	generates code for the Intel PIV family and Athlon XP uses the NASM assembler

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

`-BOEHM` Causes the program to be linked with the Boehm conservative garbage collector.

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

`-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 5.1 are supported.

5.1.3 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. For the Pentium processor it is possible to use the GNU `as` assembler instead. If NASM is being used under windows the file name for the assembler must be 'nasm.exe' rather than 'nasmw.exe'.

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. It should be noted that the cygwin version of the gcc loader does not appear to be 100% compatible with the elf format files produced by Nasm. When using Nasm under Windows the use of the DJGPP port of gcc is to be prefered. If problems arise with use of Nasm with Cygwin gcc, then the -cpugnuPentium flag should be set. This causes the built in `as` assembler to be used instead.

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

5.2 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:
```

5.3 Array representation

The maximum number of array dimensions supported in the compiler is 5.

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.

5.4 Directives

5.4.1 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.

5.4.2 Linking to external libraries

{$L libname}

This adds a directive -llibname to the gcc loader command line during linking. The $L directive should be used in the main program file. Directives placed in unit files will only be acted upon if the unit is being recompiled along with the main program.

5.4.3 Modular arithmetic

{$M+} enables modular arithmetic for pred and succ.

{$M-} disables modular arithmetic for pred and succ.

5.4.4 Include files

The directive
{$i filename}
will cause the named file to be read as source by the compiler.

5.4.5 Ifdef

Conditional compilation is supported with the directives
{$IFDEF flag}
{$IFNDEF flag}
{$ENDIF }
{$DEFINE flag}

5.5 Chinese Usage

As mentioned before Vector Pascal allows programs in the UTF-8 encoding of Unicode as source files. Since this potentially allows the use of Chinese variables and reserved words, we have defined an auxilliary table of reserved (see tables 5.2 , 5.3). An example program using these reserved words is shown in figure 5.1.

The built in char type in Pascal is represented with 16 bits in Vector Pascal. This allows any Unicode character to be handled. Chinese characters can therefore be embeded in strings and output to text files.

Chinese support was added to Vector Pascal by Gang Gao.

Figure 5.1: A program that uses Chinese Reserved Words

```
{Chin_Prog01}
{Print a Cosine graph going across the screen}
{26/11/2002}
程序 Graph(output);
   变量
     d:0..95;
      y:real;
   开始{BEGIN}
      写行; {Writeln}
      写行; {Writeln}
      写行('********COSINE GRAPTH********'); {Writeln('COSINE GRAPTH')}
      d:=0;
      当 d <= 90 循环 {while d <= 90 do}
          开始 {begin}
          y:= 余弦(d*Π/180)*30; {y:=cos(d*pi/180)*30}
          如果 四舍五入(y) = 0  {if round(y) = 0}
        那么 写行('*')        {then writeln('*'}
          否则 写行('  ':四舍五入(y), '*'); {else writeln('  ':round(y), '*'};
          d:=d+5;
       结束; {end;}
结束.{END.}
```

Table 5.2: Chinese Reserved Words

English	Chinese	Unicode
ABS	绝对值	7EDD, 5BF9, 503C
ADDR	地址	5730, 5740
AND	^	2227
ARRAY	数组	6570, 7EC4
BEGIN	开始	5F00, 59CB
BOOLEAN	布尔值	5E03, 5C14, 503C
BYTE2PIXEL	字转像	5B57, 8F6C, 50CF
CASE	个案	4E2A, 6848
CHR	字符	5B61, 7B26
CONST	常量	5E38, 91CF
COS	余弦	4F59, 5F26
DIA	判断	5224, 65AD
DIV	÷	00F7
DO	开始	5FAA, 73AF
DOWNTO	下至	4E0B, 81F3
END	结束	7ED3, 675F
ELSE	否则	5426, 5219
EXIT	退出	9000, 51FA
EXTERNAL	外部的	5916, 90E8, 7684
FALSE	错误	9534, 8BEF
FILE	文件	6587, 4EF6
FOR	当	4ECE
FUNCTION	函数	51FD, 6570
GOTO	跳转	8DF3, 8F6C
IF	如果	5982, 679C
IMPLEMENTATION	实现	5B9E, 73B0
IN	∋	
INTERFACE	接口	63A5, 53E3
LABEL	标签	6807, 7B7E
LIBRARY	库	5E93
LN	自然对数	81EA, 7136, 5BF9, 6570
MAX	最大值	6700, 5927, 503C
MIN	最小值	6700, 5C0F, 503C
MOD	%	0025
NAME	名称	540D, 79F0
NEW	新建	65B0, 5EFA
NOT	¬	00AC
OF	的	7684
OPERATOR	运算符	8FD0, 7B97, 7B26
OR	∧	2228
ORD	字转数	5B57, 8F6C, 6570
OTHERWISE	否则	5426, 6570
PACKED	封装	5C01, 88C5
PERM	排列	6392, 5217
PIXEL2BYTE	像转字	50CF, 8F6C, 5B57
POINTER	指针	6307, 9488
POW	幂	5E42

Table 5.3: Chinese Reserved Words- continued

PRED	前移	524D, 79FB
PROCEDURE	过程	8FDB, 7A0B
PROGRAM	程序	7A0B, 5E8F
PROTECTED	保护	4FDD, 62A4
READ	读	8BFB
READLN	读行	8BFB, 884C
REAL	实数	5B9E, 6570
RECORD	记录	8BB0, 5F55
REPEAT	重复	91CD, 590D
ROUND	四舍五入	56DB, 820D, 4E94, 5165
SET	集合	96C6, 5408
SHL	左位移	5DE6, 4F4D, 79FB
SHR	右位移	53F3, 4F4D, 79FB
SIN	正弦	6B63, 5F26
SIZEOF	长度	957F, 5EA6
STRING	串	4E32
SQRT	$\sqrt{}$	221A
SUCC	后移	540E, 79FB
TAN	正切	6B63, 5207
THEN	那么	90A3, 4E48
TO	到	5230
TRANS	距阵变换	8DDD, 9635, 53D8, 6362
TRUE	真	771F
TYPE	类型	7C7B, 578B
VAR	变量	53D8, 91CF
WITH	与	4E0E
WHILE	当	5F53
WRITE	写	5199
WRITELN	写行	5199, 884C
UNIT	单元	5355, 5143
UNTIL	直到	76F4, 5230
USES	使用	4F7F, 7528

Chapter 6

Compiler porting tools

Vector Pascal is an open-source project. It aims to create a productive an efficient program development environment for SIMD programming. In order to validate the concepts it has been developed initially for the Intel family of processors running Linux and Microsoft Windows. However it has been intended from the outset that the technology should be portable to other families of CPUs. This chapter addresses some of the issues involved in porting the compiler to new systems.

6.1 Dependencies

The Vector Pascal compiler tool-set can be divided along two axes as shown in figure 6.1.

1. Tools can be divided into (a) those provided as part of the release , versus (b) tools provided as part of the operating environment.
   (a) These are mainly written in Java, the exceptions being a small run-time library in C, a Pascal System unit, and several machine descriptions.
   (b) These are all available as standard under Linux, and Windows versions are freely downloadable from the web.
2. Tools can further divided into (a) those required for program preparation and documentation, (b) code translation tools, and (c) code generator preparation tools.
   (a) The program preparation tools are the VIPER IDE described in Chapter 8, along with the standard LaTeXdocument prepartion system, DVI viewers, and the TTH tool to prepare web enabled versions of Vector Pascal program descriptions.
   (b) The program translation tools are:
      i. The `ilcg.pascal` Java package which contains the Pascal compiler itself and classes to support Pascal type declarations. This carries out the first stage of code translation, from Pascal to an ILCG tree[10].
      ii. A set of machine generated code generators for CPUs such as the Pentium, the K6 etc. These carry out the second phase of code translation - into an assembler file.
      iii. The `ilcg.tree` Java package which supports the internal representation of ILCG trees (see section 6.3).
      iv. The Java system which is need to run all of the above.
      v. An assembler, which is necessary to carry out the third phase of code translation, from an assembler file to a relocatable object file.

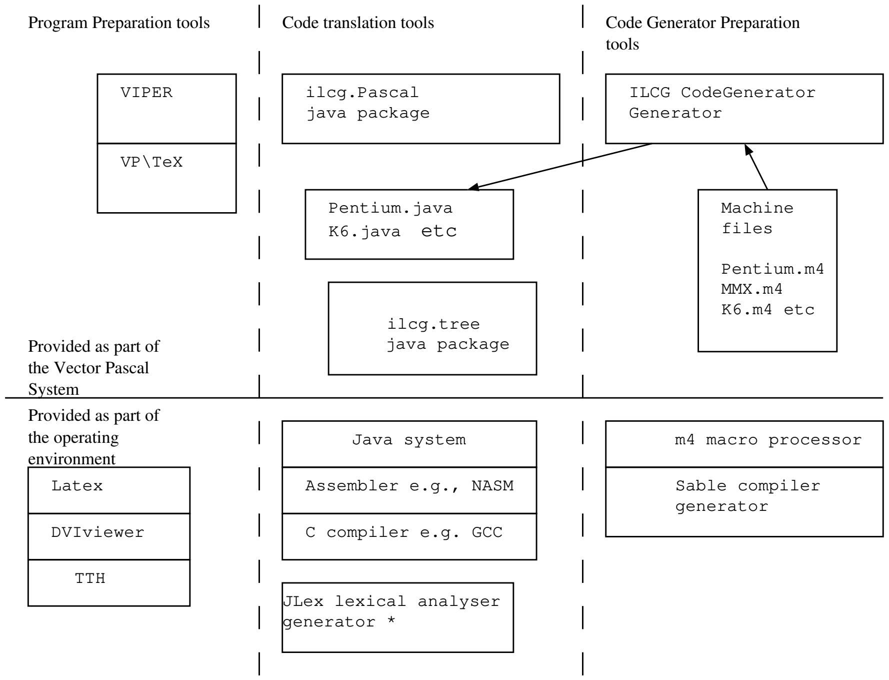

Figure 6.1: Vector Pascal toolset

vi. A C compiler and linkage system is needed to compile the C run-time library and to link the relocatable object files into final executables.

vii. In addition if one wants to alter the reserved words of Vector Pascal or make other lexical changes one needs the JLex lexical analyser generator.

6.2 Compiler Structure

The structure of the Vector Pascal translation system is shown in figure **??**. The main program class of the compiler `ilcg.Pascal.PascalCompiler.java` translates the source code of the program into an internal structure called an ILCG tree [10]. 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.

Consider first the path followed from a source file, the phases that it goes through are

i. The source file (1) is parsed by a java class PascalCompiler.class (2) a hand written, recursive descent parser[**?**], and results in a Java data structure (3), an ILCG tree, which is basically a semantic tree for the program.

ii. The resulting tree is transformed (4) from sequential to parallel form and machine independent optimisations are performed. Since ILCG trees are java objects, they can contain methods to self-optimise. Each class contains for instance a method `eval` which attempts to evaluate a tree at compile time. Another method `simplify` applies generic machine independent transpormations to the code. Thus the `simplify` method of the class `For` can perform loop unrolling, removal of redundant loops etc. Other methods allow tree walkers to apply context specific transformations.

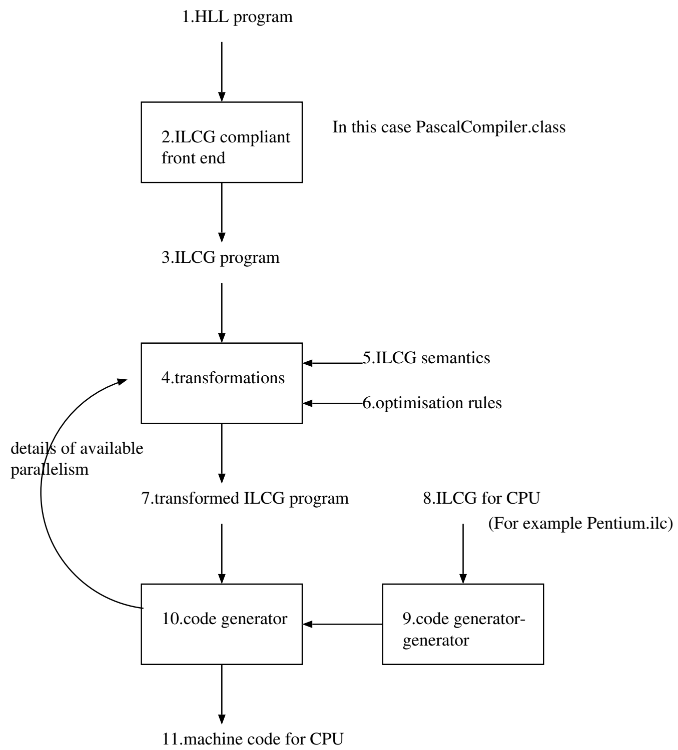

Figure 6.2: The translation of Vector Pascal to assembler.

```
{ var  i;
  for  i=1 to 9 step 1 do {
   v1[^i]:= +(^(v2[^i]),^(v3[^i]));
  };
}
```

Figure 6.3: Sequential form of array assignment

iii. The resulting ilcg tree (7) is walked over by a class that encapsulates the semantics of the target machine's instructionset (10); for example Pentium.class. During code generation the tree is futher transformed, as machine specific register optimisations are performed. The output of this process is an assembler file (11).

iv. This is then fed through an appropriate assembler and linker, assumed to be externally provided to generate an executable program.

6.2.1 Vectorisation

The parser initially generates serial code for all constructs. It then interogates the current code generator class to determine the degree of parallelism possible for the types of operations performed in a loop, and if these are greater than one, it vectorises the code.

Given the declaration
var v1,v2,v3:array[1..9] of integer;
then the statement
v1:=v2+v3;
would first be translated to the ILCG sequence shown in figure 6.3 In the example

```
{ var i;
   for i=    1 to     8 step     2 do {
    (ref int32 vector ( 2 ))mem(+(@v1,*(-(^i,1),4))):=
       +(^((ref int32 vector ( 2 ))mem(+(@v2,*(-(^i,1),4)))),
         ^((ref int32 vector ( 2 ))mem(+(@v3,*(-(^i,1),4)))));
   };
   for i=    9 to     9 step     1 do {
      v1[^i]:= +(^(v2[^i]),^(v3[^i]));
   };
}
```

Figure 6.4: Parallelised loop

above variable names such as `v1` and `i` have been used for clarity. In reality `i` would be an addressing expression like:

`(ref int32)mem(+(^((ref int32)ebp),     -1860)),`

which encodes both the type and the address of the variable. The code generator is queried as to the parallelism available on the type `int32` and, since it is a Pentium with MMX, returns 2. The loop is then split into two, a portion that can be executed in parallel and a residual sequential component, resulting in the ILCG shown in figure 6.4. In the parallel part of the code, the array subscriptions have been replaced by explictly cast memory addresses. This coerces the locations from their original types to the type required by the vectorisation. Applying the `simplify` method of the For class the following generic transformations are performed:

1. The second loop is replaced by a single statement.
2. The parallel loop is unrolled twofold.
3. The For class is replaced by a sequence of statements with explicit gotos.

The result is shown in figure 6.5. When the `eval` method is invoked, constant folding causes the loop test condition to be evaluated to

`if >(^i,8) then goto leb4af11b47f`.

6.2.2 Porting strategy

To port the compiler to a new machine, say a G5, it is necessary to

1. Write a new machine description `G5.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/G5.ilc ilcg/tree/G5.java G5`
3. Write an interface class `ilcg/tree/G5CG` which is a subclass of `G5` and which invokes the assembler and linker. The linker and assembler used will depend on the machine but one can assume that at least a `gcc` assembler and linker will be available. The class `G5CG` must take responsibility to handle the translation of procedure calls from the abstract form provided in ILCG to the concrete form required by the G5 processor.
4. The class `G5CG` should also export the method `getparallelism` which specifies to the vectoriser the degree of parallelism available for given data types. An example

```
{ var i:
  i:= 1;
  leb4af11b47e:
  if >( 2, 0) then if >(^i,8) then goto leb4af11b47f
              else null
                      fi
         else if <(^i, 8) then goto leb4af11b47f
         else null
         fi
  fi;
 (ref int32 vector ( 2 ))mem(+(@v1,*(-(^i,1),4))):=
       +(^((ref int32 vector ( 2 ))mem(+(@v2,*(-(^i,1),4)))),
         ^((ref int32 vector ( 2 ))mem(+(@v3,*(-(^i,1),4)))));
  i:=+(^i,2);
 (ref int32 vector ( 2 ))mem(+(@v1,*(-(^i,1),4))):=
       +(^((ref int32 vector ( 2 ))mem(+(@v2,*(-(^i,1),4)))),
         ^((ref int32 vector ( 2 ))mem(+(@v3,*(-(^i,1),4)))));
  i:=+(^i,2);
  goto leb4af11b47e;
  leb4af11b47f:
  i:=    9;
  v1[^i]:= +(^(v2[^i]),^(v3[^i]));
}
```

Figure 6.5: After applying `simplify` to the tree

```
 mov DWORD ecx,     1
leb4b08729615:
 cmp DWORD ecx,      8
 jg near  leb4b08729616
 lea edi,[  ecx-(     1)]; substituting in edi with 3 occurences
 movq MM1, [  ebp+edi* 4+     -1620]
 paddd MM1, [  ebp+edi* 4+     -1640]
 movq  [  ebp+edi* 4+     -1600],MM1
 lea ecx,[  ecx+     2]
 lea edi,[  ecx-(     1)]; substituting in edi with 3 occurences
 movq MM1, [  ebp+edi* 4+     -1620]
 paddd MM1, [  ebp+edi* 4+     -1640]
 movq  [  ebp+edi* 4+     -1600],MM1
 lea ecx,[  ecx+     2]
 jmp  leb4b08729615
leb4b08729616:
```

Figure 6.6: The result of matching the parallelised loop against the Pentium instruction set

```
public int getParallelism(String elementType)
{   if(elementType.equals(Node.int32)) return 2;
    if(elementType.equals(Node.int16)) return 4;
    if(elementType.equals(Node.int8)) return 8;
    if(elementType.equals(Node.uint32)) return 2;
    if(elementType.equals(Node.uint16)) return 4;
    if(elementType.equals(Node.uint8)) return 8;
    if(elementType.equals(Node.ieee32))return 4;
    if(elementType.equals(Node.ieee64))return 1;
    return 1;
}
```

Figure 6.7: The method getParallelism for a P4 processor.

> for a P4 is given in figure 6.7. Note that although a P4 is potentially capable of performing 16 way parallelism on 8 bit operands the measured speed when doing this on is less than that measured for 8 way parallelism. This is due to the restriction placed on un-aligned loads of 16 byte quantities in the P4 architecture. For image processing operations aligned accesses are the exception. Thus when specifying the degree of parallelism for a processor one should not simply give the maximal degree supported by the architecture. The maximal level of parallelism is not necessarily the fastest.

Sample machine descriptions for the Pentium and 486 are given in chapter 7 to help those wishing to port the compiler. These are given in the ILCG machine description language, an outline of which follows.

6.3 ILCG

The purpose of ILCG (Intermediate Language for Code Generation) is to mediate between CPU instruction sets and high level language programs. It poth provides a representation to which compilers can translate a variety of source level programming languages and also a notation for defining the semantics of CPU instructions.

Its purpose is to act as an input to two types of programs:

1. ILCG structures produced by a HLL compiler are input to an automatically constructed code generator, working on the syntax matching principles described in [12]. This then generates equivalent sequences of assembler statements.

2. Machine descriptions written as ILCG source files are input to code-generator-generators which produce java programs which perform function (1) above.

So far one HLL compiler producing ILCG structures as output exists: the Vector Pascal compiler. There also exists one code-generator-generator which produces code generators that use a top-down pattern matching technique analogous to Prolog unification.

ILCG is intended to be flexible enough to describe a wide variety of machine architectures. In particular it can specify both SISD and SIMD instructions and either stack-based or register-based machines. However, it does assume certain things about the machine: that certain basic types are supported and that the machine is addressed at the byte level.

In ILCG all type conversions, dereferences etc have to be made absolutely explicit.

In what follows we will designate terminals of the language in bold thus **octet** and nonterminal in sloping font thus *word8*.

6.4 Supported types

6.4.1 Data formats

The data in a memory can be distinguished initially in terms of the number of bits in the individually addressable chunks. The addressable chunks are assumed to be the powers of two from 3 to 7, so we thus have as allowed formats:*word8*, *word16*, *word32*, *word64*, *word128*. These are treated as non terminals in the grammar of ILCG.

When data is being explicitly operated on without regard to its type, we have terminals which stand for these formats: **octet, halfword, word, doubleword, quadword**.

6.4.2 Typed formats

Each of these underlying formats can contain information of different types, either signed or unsigned integers, floats etc. ILCG allows the following integer types as terminals :**int8, uint8, int16, uint16, int32, uint32, int64, uint64** to stand for signed and unsigned integers of the appropriate lengths.

The integers are logically grouped into *signed* and *unsigned*. As non-terminal types they are represented as *byte*, *short*, *integer*, *long* and *ubyte*, *ushort*, *uinteger*, *ulong*.

Floating point numbers are either assumed to be 32 bit or 64 bit with 32 bit numbers given the nonterminal symbols *float*,*double*. If we wish to specify a particular representation of floats of doubles we can use the terminals **ieee32, ieee64**.

6.4.3 Ref types

ILCG uses a simplified version of the Algol-68 reference typing model. A value can be a reference to another type. Thus an integer when used as an address of a 64 bit floating point number would be a **ref ieee64** . Ref types include registers. An integer register would be a **ref int32** when holding an integer, a **ref ref int32** when holding the address of an integer etc.

6.5 Supported operations

6.5.1 Type casts

The syntax for the type casts is C style so we have for example `(ieee64) int32` to represent a conversion of an 32 bit integer to a 64 bit real. These type casts act as constraints on the pattern matcher during code generation. They do not perform any data transformation. They are inserted into machine descritions to constrain the types of the arguments that will be matched for an instruction. They are also used by compilers to decorate ILCG trees in order both to enforce, and to allow limited breaking of, the type rules.

6.5.2 Arithmetic

The allowed dyadic arithmetic operations are addition, saturated addition, multiplication, saturated multiplication, subtraction, saturated subtraction, division and remainder with operator symboles **+, +:, *, *:, -, -:, div , mod**..

The concrete syntax is prefix with bracketing. Thus the infix operation $3 + 5 \div 7$ would be represented as **+(3 div (5 7))**.

6.5.3 Memory

Memory is explicitly represented. All accesses to memory are represented by array operations on a predefined array **mem**. Thus location 100 in memory is represented as

mem(100). The type of such an expression is *address*. It can be cast to a reference type of a given format. Thus we could have

(ref int32)mem(100)

6.5.4 Assignment

We have a set of storage operators corresponding to the word lengths supported. These have the form of infix operators. The size of the store being performed depends on the size of the right hand side. A valid storage statement might be

(ref octet)mem(299) :=(int8) 99

The first argument is always a reference and the second argument a value of the appropriate format.

If the left hand side is a format the right hand side must be a value of the appropriate size. If the left hand side is an explicit type rather than a format, the right hand side must have the same type.

6.5.5 Dereferencing

Dereferencing is done explicitly when a value other than a literal is required. There is a dereference operator, which converts a reference into the value that it references. A valid load expression might be:

(octet)↑ ((ref octet)mem(99))

The argument to the load operator must be a reference.

6.6 Machine description

Ilcg can be used to describe the semantics of machine instructions. A machine description typically consists of a set of register declarations followed by a set of instruction formats and a set of operations. This approach works well only with machines that have an orthogonal instruction set, ie, those that allow addressing modes and operators to be combined in an independent manner.

6.6.1 Registers

When entering machine descriptions in ilcg registers can be declared along with their type hence

register word EBX assembles['ebx'] ;
reserved register word ESP assembles['esp'];

would declare **EBX** to be of type **ref word**.

Aliasing

A register can be declared to be a sub-field of another register, hence we could write

alias register octet AL = EAX(0:7) assembles['al'];
alias register octet BL = EBX(0:7) assembles['bl'];

to indicate that **BL** occupies the bottom 8 bits of register **EBX**. In this notation bit zero is taken to be the least significant bit of a value. There are assumed to be two pregiven registers **FP, GP** that are used by compilers to point to areas of memory. These can be aliased to a particular real register.

register word EBP assembles['ebp'] ;
alias register word FP = EBP(0:31) assembles ['ebp'];

Additional registers may be reserved, indicating that the code generator must not use them to hold temporary values:

reserved register word ESP assembles['esp'];

6.6.2 Register sets

A set of registers that are used in the same way by the instructionset can be defined.

pattern reg means [$EBP|EBX|ESI|EDI|ECX|EAX|EDX|ESP$**] ;**

pattern breg means[$AL|AH|BL|BH|CL|CH|DL|DH$**];**

All registers in an register set should be of the same length.

6.6.3 Register Arrays

Some machine designs have regular arrays of registers. Rather than have these exhaustively enumerated it is convenient to have a means of providing an array of registers. This can be declared as:

register vector(8)doubleword MM assembles['MM'i] ;

This declares the symbol MMX to stand for the entire MMX register set. It implicitly defines how the register names are to be printed in the assembly language by defining an indexing variable i that is used in the assembly language definition.

We also need a syntax for explicitly identifying individual registers in the set. This is done by using the dyadic subscript operator:

subscript(MM,2)

which would be of type **ref doubleword**.

6.6.4 Register Stacks

Whilst some machines have registers organised as an array, another class of machines, those oriented around postfix instructionsets, have register stacks.

The ilcg syntax allows register stacks to be declared:

register stack (8)ieee64 FP assembles[' '] ;

Two access operations are supported on stacks:

PUSH is a void dyadic operator taking a stack of type ref t as first argument and a value of type t as the second argument. Thus we might have:

PUSH(FP,↑mem(20))

POP is a monadic operator returning t on stacks of type t. So we might have

mem(20):=POP(FP) In addition there are two predicates on stacks that can be used in pattern pre-conditions.

FULL is a monadic boolean operator on stacks.

EMPTY is a monadic boolean operator on stacks.

6.6.5 Instruction formats

An instruction format is an abstraction over a class of concrete instructions. It abstracts over particular operations and types thereof whilst specifying how arguments can be combined.

instruction pattern
RR(operator op, anyreg r1, anyreg r2, int t)
means[r1:=(t) op(↑((ref t) r1),↑((ref t) r2))]
assembles[op ' ' r1 ',' r2];

In the above example, we specify a register to register instruction format that uses the first register as a source and a destination whilst the second register is only a destination. The result is returned in register r1.

We might however wish to have a more powerful abstraction, which was capable of taking more abstract apecifications for its arguments. For example, many machines allow arguments to instructions to be addressing modes that can be either registers or memory references. For us to be able to specify this in an instruction format we need to be able to provide grammer non-terminals as arguments to the instruction formats.

For example we might want to be able to say

instruction pattern
RRM(operator op, reg r1, maddrmode rm, int t)
means [r1:=(t) op(↑((ref t)r1),↑((ref t) rm))]
assembles[op ' ' r1 ',' rm] ;

This implies that addrmode and reg must be non terminals. Since the non terminals required by different machines will vary, there must be a means of declaring such non-terminals in ilcg.

An example would be:

pattern regindirf(reg r)
means[↑(r)] assembles[r];
pattern baseplusoffsetf(reg r, signed s)
means[+(↑(r) ,const s)] assembles[r '+' s];
pattern addrform means[baseplusoffsetf| regindirf];
pattern maddrmode(addrform f)
means[mem(f)] assembles['[' f ']'];

This gives us a way of including non terminals as parameters to patterns.

6.7 Grammar of ILCG

The following grammar is given in Sable [34] compatible form. The Sable parser generator is used to generate a parser for ILCG from this grammar. The ILCG parser then translates a CPU specification into a tree structure which is then walked by an ILCG-tree-walk-generator to produce an ILCG-tree-walk Java class specific to that CPU.

If the ILCG grammar is extended, for example to allow new arithmetic operators, then the ILCG-tree-walk-generator must itself be modified to generate translation rules for the new operators.

/*

6.8 ILCG grammar

This is a definition of the grammer of ILCG using the Sable grammar specification lanaguage. It is input to Sable to generate a parser for machine descriptions in ilcg

```
*/



Package ilcg;
/*
```

6.8.1 Helpers

Helpers are regular expressions macros used in the definition of terminal symbols of the grammar.

```
*/
Helpers
  letter = [['A'..'Z']+['a'..'z']];
  digit = ['0'..'9'];
```

```
  alphanum = [letter+['0'..'9']];
  cr = 13;
  lf = 10;
  tab = 9;
    digit_sequence = digit+;
    fractional_constant = digit_sequence? '.' digit_sequence | digit_sequence '.';
    sign = '+' | '-';
    exponent_part = ('e' | 'E') sign? digit_sequence;
    floating_suffix = 'f' | 'F' | 'l' | 'L';
  eol = cr lf | cr | lf;        // This takes care of different platforms
  not_cr_lf = [[32..127] - [cr + lf]];
  exponent = ('e'|'E');
  quote = ''';
  all =[0..127];
  schar = [all-'''];
  not_star = [all - '*'];
  not_star_slash = [not_star - '/'];
/*
```

6.8.2 Tokens

The tokens section defines the terminal symbols of the grammar.

```
*/
Tokens
  floating_constant = fractional_constant exponent_part? floating_suffix? |
        digit_sequence exponent_part floating_suffix?;
 /*
```

terminals specifying data formats

```
*/
 void ='void';
 octet = 'octet'; int8 = 'int8'; uint8 = 'uint8';
 halfword = 'halfword'; int16 = 'int16' ; uint16 = 'uint16' ;
 word = 'word'; int32 = 'int32' ;
 uint32 = 'uint32' ; ieee32 = 'ieee32';
 doubleword = 'doubleword'; int64 = 'int64' ;
 uint64 = 'uint64'; ieee64 = 'ieee64';
 quadword = 'quadword';
/*
```

terminals describing reserved words

```
*/
 function= 'function';
 flag = 'flag';
 location = 'loc';
 procedure='instruction';
 returns ='returns';
 label = 'label';
 goto='goto';
 fail ='interrupt';
 for ='for';
 to='to';
 step='step';
 do ='do';
 ref='ref';
 const='const';
 reg= 'register';
```

```
 operation = 'operation';
 alias = 'alias';
 instruction = 'instruction';
 address = 'address';
 vector ='vector';
 stack = 'stack';
 sideeffect='sideeffect';
 if ='if';
 reserved='reserved';
 precondition ='precondition';

 instructionset='instructionset';
/*
```

terminals for describing new patterns

```
 */
 pattern = 'pattern';
 means = 'means';
 assembles = 'assembles';

/*
```

terminals specifying operators

```
*/
 colon = ':';
 semicolon= ';';
 comma = ',';
 dot = '.' ;
 bra ='(';

 ket =')';
 plus = '+';
 satplus = '+:';
 satminus = '-:';
 satmult ='*:';
 map='->';
 equals = '=';
 le = '<=';
 ge='>=';
 ne='<>';
 shl='<<';
 shr='>>';
 lt='<';
 gt='>';
 minus = '-';
 times = '*';
 exponentiate = '**';
 divide = 'div';
 replicate = 'rep';
 and = 'AND';
 or = 'OR' ;
 xor = 'XOR';
 not = 'NOT';
 sin='SIN';
 cos='COS';
 abs='ABS';
 tan='TAN';
 ln='LN';
```

```
 min='MIN';
 max='MAX';
 sqrt='SQRT';
 trunc='TRUNCATE';
 round='ROUND';
 float='FLOAT';
 remainder = 'MOD';
 extend= 'EXTEND';
 store = ':=';
 deref = '^';
 push ='PUSH';
 pop ='POP';
 call='APPLY';
 full='FULL';
 empty='EMPTY';
 subscript='SUBSCRIPT';
 intlit = digit+;

 vbar = '|';
 sket=']';
 sbra='[';
 end='end';
 typetoken='type';
 mem='mem';
 string = quote schar+ quote;
/*
```

identifiers come after reserved words in the grammar

```
*/
 identifier = letter alphanum*;
 blank = (' '|cr|lf|tab)+;
 comment = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';

Ignored Tokens
blank,comment;
/*
```

6.8.3 Non terminal symbols

```
*/
Productions
 program = statementlist instructionlist;
 instructionlist =instructionset sbra alternatives sket;
/*
```

non terminals specifying data formats

```
*/
 format = {octet} octet| {halfword} halfword |
          {word} word | {doubleword} doubleword |
          {quadword} quadword;


/*
```

non terminals corresponding to type descriptions

```
*/
```

```
 reference =   ref type ;
 array = vector bra number ket;
 aggregate={stack} stack bra number ket |{vector}array |{non};
 predeclaredtype =   {format} format|{tformat}tformat ;
 typeprim= {typeid}  typeid| {predeclaredtype}predeclaredtype;
type=  {predeclaredtype}predeclaredtype|
{typeid}  typeid|
{array}typeprim array|
  {cartesian}sbra type cartesian* sket|
{reftype}reference|
{map}bra [arg]:type map [result]:type ket;
 cartesian =  comma type;

 tformat = {signed} signed|{unsigned}unsigned|{ieee32}ieee32|{ieee63}ieee64;
 signed = int32 | {int8} int8 | {int16} int16 | {int64} int64;
 unsigned = uint32 | {uint8} uint8 | {uint16} uint16 |
            {uint64} uint64;


/*
```

non terminals corresponding to typed values

```
*/
 value = /*{refval}refval |      */
  {rhs}rhs|
  {loc}loc|
  {void}void|
  {cartval}cartval|
{dyadic} dyadic bra [left]:value comma [right]:value ket|
{monadic}monadic bra value ket;
/*
```

value corresponding to a cartesian product type e.g. record initialisers

```
*/
 cartval=sbra value carttail* sket;
 carttail = comma value;
/*
```

conditions used in defining control structures

```
*/
condition={dyadic} dyadic bra [left]:condition comma [right]:condition ket|
{monadic}monadic bra condition ket |
{id}identifier|
{number}number;
 rhs= {number}number|
  {cast}bra type ket value|
  {const}const identifier |
   {deref}deref  bra  refval ket;

 refval = loc|
  {refcast} bra type ket loc;
 loc =  {id}identifier|
  {memory}mem bra value ket ;


/*predeclaredregister = {fp}fp|{gp}gp;*/
 number =  {reallit} optionalsign reallit|
```

```
           {integer} optionalsign intlit;
 optionalsign = |{plus}plus|{minus}minus;
 reallit= floating_constant;
/*
```

operators

```
*/
dyadic  = {plus} plus|
{minus} minus |
{identifier} identifier|
{exp}exponentiate|
  {times} times |
  {divide}  divide|
{replicate} replicate|
  {lt}lt|
  {gt}gt|
  {call}call|
{le}le|
{ge}ge|
{eq}equals|
{ne}ne|
{min}min|{max}max|
{push}push|
{subscript}subscript|
{satplus}satplus|
{satmult}satmult|
{satminus}satminus|
{shl}shl|
{shr}shr|
              {remainder} remainder|
              {or}or|
              {and}and|
              {xor}xor;
monadic={not}not|{full}full|{empty}empty|{pop}pop|{sin}sin|
{trunc}trunc|{round}round|{float}float| {extend}extend|
{cos}cos|{tan}tan|{abs}abs|{sqrt}sqrt |{ln}ln;
/*
```

register declaration

```
*/
registerdecl= reservation reg aggregate type identifier assembles sbra string sket  ;
reservation = {reserved}reserved|{unreserved};

aliasdecl =
alias reg aggregate type
          [child]:identifier equals [parent]:identifier bra [lowbit]:intlit colon [highbit]:intlit ket
          assembles sbra string sket;

opdecl = operation identifier means operator assembles sbra string sket;
operator =  {plus}plus|
{minus}minus|
{times}times|
{lt}lt|
{gt}gt|
{min}min|
{max}max|
{shl}shl|
{shr}shr|
```

```
  {le}le|
{ge}ge|
{eq}equals|
{ne}ne|
{divide} divide|
            {remainder}remainder|
{or}or|
{and}and|
{xor}xor;
```

/*

pattern declarations

```
*/
assign =  refval store value ;
meaning = {value}value|
{assign}assign|
{goto}goto value|
{fail}fail value|
{if}if bra value ket meaning|
{for} for refval store [start]:value to [stop]:value step [increment]:value do meaning|
   {loc}location value;
patterndecl = pattern identifier paramlist means sbra meaning sket assemblesto sideeffects precond
               |
              {alternatives} pattern identifier means sbra alternatives sket;

paramlist = bra param paramtail* ket|{nullparam}bra ket;
param = typeid identifier|{typeparam} typetoken identifier|{label}label identifier;
typeid = identifier;
paramtail = comma param;
alternatives = type alts*;
alts = vbar type;
precond = precondition sbra condition sket|{unconditional};
asideeffect=   sideeffect  returnval;
sideeffects = asideeffect*;
assemblesto=assembles sbra assemblypattern sket;
assemblypattern = assemblertoken*;
assemblertoken = {string} string | {identifier} identifier;
returnval = returns identifier;
/*
```

statements

```
*/
statement = {aliasdecl} aliasdecl|
            {registerdecl} registerdecl |
{addressmode} address patterndecl|
{instructionformat}procedure patterndecl|
{opdecl}opdecl|
{flag} flag identifier equals intlit|
{typerename}typetoken predeclaredtype     equals identifier|
{patterndecl} patterndecl;
statementlist = statement semicolon statements*;
statements =  statement  semicolon;


//
```

Chapter 7

Sample Machine Descriptions

7.1 Basic 386 architecture

/*

Basic ia32 processor description int ilcg copyright(c) Paul Cockshott, University of Glasgow Feb 2000

7.1.1 Declare types to correspond to internal ilcg types

```
*/

type word=DWORD;
type uint32=DWORD;
type int32=DWORD;
type ieee64=QWORD;
type int64=QWORD;
type octet=BYTE;
type uint8=BYTE;
type int16=WORD;
type uint16=WORD;
type int8=BYTE;
type ieee32=DWORD;
type halfword=WORD;

/*
```

7.1.2 compiler configuration flags

```
*/
flag realLitSupported = 0;
/*
```

7.1.3 Register declarations

```
*/
register int32 EAX assembles['eax'] ;
register int32 ECX assembles['ecx'] ;
register int32 EBX assembles['ebx'] ;
register word EBP assembles['ebp'] ;
alias register word FP = EBP(0:31) assembles ['ebp'];
reserved register word ESP assembles['esp'];
register int32 ESI assembles['esi'] ;
register int32 EDI assembles['edi'] ;
```

```
register int32 EDX assembles['edx'];
/*register int32 fitemp assembles['dword[fitemp]'];/* not a real register */
alias register uint32 uax= EAX (0:31) assembles ['eax'] ;
alias register uint32 ucx= ECX (0:31) assembles ['ecx'] ;
alias register uint32 ubx= EBX (0:31) assembles ['ebx'] ;
alias register uint32 usi= ESI (0:31) assembles ['esi'] ;
alias register uint32 udi= EDI (0:31) assembles ['edi'] ;
alias register uint32 udx= EDX (0:31) assembles ['edx'];



/* use these for signed 8 bit values */
 alias register int8 AL = EAX(0:7) assembles['al'];
 alias register int8 BL = EBX(0:7) assembles['bl'];
 alias register int8 CL = ECX(0:7) assembles['cl'];
 alias register int8 DL = EDX(0:7) assembles['dl'];
 alias register int8 iBH = EBX(8:15) assembles['bh'];
 alias register int8 iCH = ECX(8:15) assembles['ch'];
 alias register int8 iDH = EDX(8:15) assembles['dh'];
/* use these for unsigned 8 bit values */
/* alias register uint8 AH = EAX(8:15) assembles['ah']; dont use this*/
 alias register uint8 BH = EBX(8:15) assembles['bh'];
 alias register uint8 CH = ECX(8:15) assembles['ch'];
 alias register uint8 DH = EDX(8:15) assembles['dh'];
 alias register uint8 uAL = EAX(0:7) assembles['al'];
 alias register uint8 uBL = EBX(0:7) assembles['bl'];
 alias register uint8 uCL = ECX(0:7) assembles['cl'];
 alias register uint8 uDL = EDX(0:7) assembles['dl'];

/* use these for untyped 8 bit values */
 alias register octet oAL = EAX(0:7) assembles['al'];
 alias register octet oBL = EBX(0:7) assembles['bl'];
 alias register octet oCL = ECX(0:7) assembles['cl'];
 alias register octet oDL = EDX(0:7) assembles['dl'];

 alias register int16 AX =EAX(0:15)assembles['ax'];
 alias register int16 BX =EBX(0:15)assembles['bx'];

 alias register int16 DX =EDX(0:15)assembles['dx'];
 alias register int16 CX =ECX(0:15)assembles['cx'];
 alias register uint16 uAX =EAX(0:15)assembles['ax'];
 alias register uint16 uBX =EBX(0:15)assembles['bx'];

 alias register uint16 uDX =EDX(0:15)assembles['dx'];
 alias register uint16 uCX =ECX(0:15)assembles['cx'];
 alias register halfword SI = ESI(0:15)assembles['si'];
 alias register halfword DI = EDI(0:15)assembles['di'];

register stack(4096)int32 mainSTACK assembles[ 'mainSTACK'];
                  /*
```

7.1.4 Register sets

There are several intersecting sets of registers defined for different instructions. Note that the ECX and CL,CH registers are named last in their lists to increase the chance that they are free for sepecial instructions that need them.

```
*/
pattern indexreg means[EDI|ESI|EBX|EBP|ESP|EAX|EDX|ECX];
```

```
pattern accumulators means[EAX|EDX|ECX|EBX];
pattern ireg means [  indexreg]  ;
pattern ureg means [EBP| ubx|udi|usi|udx|ESP|ucx|uax ]  ;

pattern reg means [ireg|ureg];

/* Note that the order of the byte registers is chosen to keep the ah and al regs
   free for instructions that require themspecifically, particularly
conditional expressions on the floating point stack, that return boolean
results in al */
pattern bireg means[ BL|DL|AL|CL|iBH|iDH|iCH];
pattern bureg means[BH|DH|uAL|uBL|uDL|uCL|CH];
pattern boreg means[oBL|oAL|oDL|oCL];
pattern bacc means[AL];
pattern bnonacc means[BL|CL|DL];
pattern breg means[bacc|bireg|bnonacc|bureg|boreg];
pattern swreg means[BX|CX|DX|AX];
pattern uwreg means[uBX|uCX|uDX];
pattern untypedwreg means[SI|DI];
pattern wreg means[swreg|uwreg|untypedwreg];
pattern pushreg means[reg|wreg]; /* these are directly pushable */
/*pattern dummyreg means[fitemp];*/
pattern dpushreg means[pushreg];
pattern anyreg means[ breg|wreg|reg];
pattern signedreg means[bireg|ireg|wreg];
pattern unsignedreg means[bureg|ureg|wreg];
pattern acc means[EAX];

pattern dacc means[EDX];
pattern wacc means[AX];
pattern ebxacc means[EBX];
pattern ebxbacc means[BL];
pattern ecxacc means[ECX];
pattern ecxbacc means[CL];
pattern ecxuacc means[ucx];
pattern modreg means [ECX];
pattern sourcereg means [ESI];
pattern destreg means [EDI];
pattern countreg means [ECX];
pattern shiftcountreg means [ecxbacc|ecxacc|ecxuacc];

/*
```

7.1.5 Operator definition

This section defines operations that can be used to parameterise functions.

```
*/
operation add means  +  assembles [ 'add'];
/* */operation and means  AND  assembles[ 'and'];
operation or means OR assembles['or'];
operation xor means XOR assembles['xor'];/* */
operation sub means  -  assembles [ 'sub'];
operation mul means * assembles ['mul'];
operation imul means * assembles ['imul '];
operation bel means < assembles ['b'];
operation lt means < assembles ['l'];
```

```
operation ab means > assembles ['a'];
operation gt means > assembles ['g'];

operation eq means = assembles ['z'];

operation be means <= assembles ['be'];
operation le means <= assembles ['le'];

operation ae means >= assembles ['ae'];
operation ge means >= assembles ['ge'];
operation ne means <> assembles ['nz'];
operation shiftleft means << assembles ['l'];
operation shiftright means >> assembles ['r'];

pattern condition means[ne|ge|le|eq|gt|lt];
pattern equals means[eq];
pattern unsignedcondition means[ne|ae|be|eq|ab|bel];
pattern operator means[add | sub|imul|and|or|xor];
pattern nonmultoperator means[add|sub|and|or|xor];
pattern saddoperator means[add|imul|and|or|xor];
pattern shiftop means [shiftleft|shiftright];

/*
```

7.1.6 Data formats

Here we define ilcg symbols for the types that can be used as part of instructions.

```
*/
pattern unsigned means[uint32|uint8|uint16];
pattern signed means[   int8 |   int16|int32 ];
pattern int means[   int8 |   int16 |int32| uint32|uint8|uint16];
pattern double means[ieee64] ;
pattern float  means[ieee32];
pattern real means [ieee64|float];
pattern byte means[uint8|int8|octet];

pattern word32 means[int32|uint32|word];
pattern word16 means[int16|uint16|halfword];
pattern wordupto32 means[byte|word16|word32];
pattern dataformat means[octet|word];
pattern longint means [int32|uint32];
pattern hiint means[int32|int64|int16];
pattern two(type t)means[2] assembles['2'];
pattern four(type t)means[4] assembles['4'];
pattern eight(type t)means[8] assembles['8'];

pattern scale means[two|four|eight];

/*
```

Define the address forms used in lea instructions these differ from the address forms used in other instructions as the semantics includes no memory reference. Also of course register and immediate modes are not present.

```
 */
pattern labelf(label l)
means [l]
```

```
assembles[l];
pattern sconst(signed s)means[const s]assembles[s];
pattern lconstf means[sconst|labelf];
pattern labelconstf(lconstf l,lconstf s)
means [+(l,const s)]
assembles[l'+'s];
pattern constf(signed s)
means[const s]
assembles [s];
pattern offset means[constf|labelf|labelconstf];
pattern regindirf(reg r)
means[^(r) ]
assembles[ r ];
pattern simplescaled(reg r1,scale s)
means[*(^(r1),s)]
assembles[r1 '*'s];

pattern negcompscaled(reg r1,scale s,offset o)
means[*(-(^(r1),o),s)]
assembles[r1 '*'s'-(' s'*'o')'];
pattern compscaled(reg r1,scale s,offset o)
means[*(+(^(r1),o),s)]
assembles[r1 '*'s'+(' s'*'o')'];
pattern scaled means[compscaled|negcompscaled|simplescaled];
pattern baseminusoffsetf(reg r, offset s )
means[-(   ^(r) ,  s)]
assembles[  r '-(' s ')'];
pattern baseplusoffsetf(reg r, offset s )
means[+(   ^(r) ,  s)]
assembles[  r '+' s ];
pattern scaledIndexPlusOffsetf( scaled s, offset offs)
means[+(s, offs)]
assembles[ s '+' offs];
address pattern basePlusScaledIndexf(reg r1,scaled s)
means[+(^(r1),s)]
        assembles[  r1 '+'   s   ];
address pattern basePlusScaledIndexPlusOffsetf(reg r1,scaled s,offset off,longint t)
means[+((t)+(^(r1),  off),s )]
        assembles[  r1 '+' s '+'off  ];
address pattern basePlusIndexPlusOffsetf(reg r1,reg r2,offset off)
means[+(^(r1),+(^(r2),  off))]
        assembles[  r1 '+' r2 ' +'off  ];
address pattern basePlusIndexf(reg r1,reg r2)
means [+(^(r1),^(r2))]
assembles[  r1 '+' r2   ];
pattern directf(unsigned s)
means[const s]
assembles[  s  ];


/*
```

7.1.7 Choice of effective address

This contains the useful formats for the load effective address instruction. The pattern regindirf is excluded here as it adds nothing we do not have already from mov instructions.

```
*/
pattern uncasteaform means[directf  |
```

```
labelf| labelconstf|
scaledIndexPlusOffsetf|
basePlusScaledIndexPlusOffsetf|
baseplusoffsetf |
basePlusIndexPlusOffsetf|
basePlusScaledIndexf|
        scaledIndexPlusOffsetf|
baseminusoffsetf
|basePlusIndexf
];
pattern eaform(uncasteaform f,longint t)  /* allow the address expression to be cast to an integer */
means[(t)f]
assembles[f];
/*
```

7.1.8 Formats for all memory addresses

```
*/
pattern addrform means[eaform|regindirf];


/**
```

define the address patterns used in other instructions

```
*/
```

```
pattern maddrmode(addrform f)
means[mem(f) ]
assembles[ '[' f ']' ];
pattern gmaddrmode means[/*dummyreg|*/maddrmode];
pattern immediate(signed s)means [const s] assembles [s];
pattern intimmediate(int s)means [const s] assembles [s];
pattern jumpmode means[labelf|maddrmode];
pattern addrmode means[maddrmode|anyreg];
pattern baddrmode means[maddrmode|breg];
pattern waddrmode means[maddrmode|reg];
pattern wmemdummy means[maddrmode/*|dummyreg*/];
pattern regshift(shiftcountreg r)means[^(r)] assembles['cl'];
pattern shiftcount means[immediate|regshift];
pattern regaddrop(addrmode r)means[^(r)] assembles[r];
pattern regaddrimmediate means[intimmediate|maddrmode|regaddrop/*|dummyreg*/];
/*
```

7.1.9 Instruction patterns for the 386

Stack operations

```
*/

instruction pattern STACKSTORE(reg r1 )
means[(ref int32)mem(POP(mainSTACK)):=^(r1)]
assembles['xchg DWORD[esp],'r1'\n  pop DWORD['r1']\n   '];
instruction pattern STACKWSTORE(wreg r1 )
```

```
means[(ref halfword)mem(POP(mainSTACK)):=^(r1)]
assembles['xchg DWORD[esp],esi\n  mov word[esi],'r1'\n  pop esi'];
instruction pattern STACKBSTORE(breg r1 )
means[(ref octet)mem(POP(mainSTACK)):=^(r1)]
assembles['xchg DWORD[esp],esi\n  mov BYTE[esi],'r1'\n  pop esi'];
instruction  pattern SMLIT( nonmultoperator op,offset s)
        means[ PUSH(mainSTACK,(int32)op( POP(mainSTACK), s))]
     assembles[op '  DWORD[esp] ,' s];

instruction  pattern SMULIT( nonmultoperator op,offset s)
        means[ PUSH(mainSTACK,(int32)*( POP(mainSTACK), s))]
     assembles['xchg eax,DWORD[esp]\n imul eax ,' s'\n xchg eax,DWORD[esp]' ];

instruction pattern SADD(saddoperator op)
means [PUSH(mainSTACK,(longint)+(POP(mainSTACK),POP(mainSTACK)))]
assembles['xchg eax,DWORD[esp]\n add DWORD[esp+4],eax\n pop eax'];

instruction pattern SOP(saddoperator op)
means [PUSH(mainSTACK,(longint)op(POP(mainSTACK),POP(mainSTACK)))]
assembles['xchg eax,DWORD[esp]\n 'op' eax,DWORD[esp+4]\n mov DWORD[esp+4],eax\n pop eax'];
instruction  pattern SMR( nonmultoperator op,reg r1)
        means[ PUSH(mainSTACK,(int32)op( POP(mainSTACK),^( r1)))]
     assembles[op '  DWORD[esp] ,' r1];
instruction  pattern SMRSHIFT( shiftop op,shiftcountreg r1)
        means[ PUSH(mainSTACK, op( POP(mainSTACK),^( r1)))]
     assembles['xchg eax, [esp]\n'
                 'sh'op '  eax ,cl'
                 '\n xchg eax,[esp]'];
instruction  pattern BSMR( nonmultoperator op,breg r1)
        means[ PUSH(mainSTACK,(octet)op( POP(mainSTACK),^( r1)))]
     assembles[op '  byte[esp] ,' r1];

instruction  pattern SMRADD( reg r1)
        means[ r1:=(int32)+( POP(mainSTACK),^( r1))]
     assembles['add 'r1',DWORD[esp] \n add esp,4'];

instruction  pattern SMRP( nonmultoperator op,reg r1,type t)
        means[ PUSH(mainSTACK,(ref t)op( POP(mainSTACK),^(r1)))]
     assembles[op '  DWORD[esp] ,' r1];
instruction pattern RPUSH(dpushreg r)
means[PUSH(mainSTACK,^(r))]
assembles['push '  r];
instruction pattern RPUSHE(ureg r)
means[PUSH(mainSTACK,(int64)EXTEND(^(r)))]
assembles[' push dword 0 ; extend 'r' to 64'
                 '\n push '  r];

instruction pattern RPOP(dpushreg r,type t)
means[(ref t)r:=(t)POP(mainSTACK)]
assembles['pop  '  r];
instruction pattern BPUSH(bureg r)
means[PUSH(mainSTACK,^(r))]
assembles['push dword 0\n mov BYTE[esp],'r];
instruction pattern BSPUSH(baddrmode r)
means[PUSH(mainSTACK,(int8)^(r))]
assembles['push esi\n movsx esi,'r'\n xchg esi,[esp]'];
instruction pattern BSPOP(bireg r)
means[r:=POP(mainSTACK)]
```

```
assembles['mov  '  r',BYTE[esp]\n add esp,4'];
instruction pattern BPOP(bureg r)
means[r:=POP(mainSTACK)]
assembles['mov  '  r',BYTE[esp]\n add esp,4'];


instruction pattern REFPOP(addrmode r,type t)
means[(ref ref t)r:=POP(mainSTACK)]
assembles['pop DWORD '  r];
instruction pattern MEMPOP(maddrmode m)
means[(ref int32)m:=POP(mainSTACK)]
assembles['pop DWORD 'm];
instruction pattern LITPUSH(offset s)
means[PUSH(mainSTACK, s)]
assembles['push DWORD '  s];
instruction pattern MEMPUSH(maddrmode m)
means[PUSH(mainSTACK,(word)^( m))]
assembles['push DWORD ' m];
instruction pattern DMEMPUSH(eaform ea)
means[PUSH(mainSTACK,(doubleword)^((ref doubleword)mem(ea)))]
assembles['push DWORD['ea'+4]\n push DWORD['ea']'];

instruction pattern STACKLOAD(word32 t)
means[PUSH(mainSTACK,^((ref t)mem(POP(mainSTACK))))]
assembles['xchg DWORD[esp],eax\n  mov eax,DWORD[eax]\n  xchg DWORD[esp],eax'];
instruction pattern REFPUSH(maddrmode m,type t)
means[PUSH(mainSTACK,(ref t)^(m))]
assembles['push DWORD ' m];
instruction pattern SDEREF(int t)
 means[PUSH(mainSTACK,(t)^(mem(POP(mainSTACK))))]
 assembles['xchg esi,[esp]\n mov esi,dword[esi]\n  xchg esi,[esp]'];
instruction pattern SDEREFDOUBLEWORD(int t)
 means[PUSH(mainSTACK,(doubleword)^(mem(POP(mainSTACK))))]
 assembles['xchg esi,[esp]\n push dword[esi]\n mov esi,dword[esi+4]\n  xchg esi,[esp]'];

/*
```

Data movement to and from registers

```
*/
instruction pattern SELECT(reg r1,reg r2,addrmode r3,wordupto32 t)
means[(ref t) r1:=OR(AND((t)^(r1),(t)^(r2)),AND((t)^(r3),NOT(^(r2))))]
assembles[
'and 'r1      ','      r2         '\n'
                'not 'r2                        '\n'
'and 'r2      ','   t ' '  r3         '\n'
'or 'r1       ','      r2];
instruction  pattern LOAD(maddrmode rm, anyreg r1, wordupto32 t)
        means[ (ref t) r1:= (t)^(rm )]
assembles['mov ' r1 ',' t ' ' rm];
instruction pattern MOVZXB(reg r1, baddrmode rm)
means[  r1:=(uint32)EXTEND( (uint8)^( rm) )]
assembles['movzx ' r1 ', BYTE  'rm];
instruction pattern MOVZXB2(reg r1, baddrmode rm)
means[  r1:=(int32)EXTEND( (uint8)^( rm) )]
assembles['movzx ' r1 ', BYTE  'rm];

instruction pattern MOVSXB(reg r1,baddrmode rm)
means[r1:=(int32)EXTEND( (int8)^( rm) )]
```

```
assembles['movsx 'r1',BYTE   'rm];
instruction pattern MOVZXBW(uwreg r1, breg rm)
means[  r1:= EXTEND(^(rm))]
assembles['movzx ' r1 ',   'rm];
instruction pattern MOVSXBW(swreg r1, breg rm)
means[  r1:= EXTEND(^(rm))]
assembles['movsx ' r1 ',   'rm];
instruction pattern MOVZXW(reg r1, wreg rm)
means[  r1:=(uint32)EXTEND(^(rm))]
assembles['movzx ' r1 ',   'rm];
instruction pattern MOVSXW(reg r1, wreg rm)
means[  r1:=(int32)EXTEND(^(rm))]
assembles['movsx ' r1 ',   'rm];
instruction pattern ToBYTE(reg r, breg b)
means[b:= (octet) ^( r)]
        assembles['push ' r '\nmov ' b ',BYTE[esp]\nadd esp,4 ' ];



instruction pattern STOREBR(baddrmode rm, breg r1)
        means[  (ref octet ) rm:= ^(r1) ]
assembles['mov BYTE 'rm',' r1];
instruction pattern STORER(maddrmode rm, reg r1, word32 t)
        means[  (ref t) rm:= ^( r1) ]
assembles['mov ' t ' 'rm',' r1];
 instruction pattern STOREWR(maddrmode rm, wreg r1, word16 t)
        means[  (ref t) rm:= ^( r1) ]
assembles['mov ' t ' 'rm',' r1];
instruction  pattern NULMOV(reg r3, type t)
means[(ref t)r3:=^((ref t)r3)]
        assembles[';nulmov ' r3  r3];
instruction  pattern STORELIT(addrmode rm,  type t, int s)
        means[ (ref t) rm:= (t)const s  ]
assembles['mov ' t ' 'rm ','  ' ' s];
instruction  pattern CLEARREG(reg rm,  type t, int s)
        means[ (ref t) rm:= (t)0  ]
assembles['xor '  rm ','  rm];
/*
```

Register to register arithmetic

```
*/
instruction  pattern RMLIT(nonmultoperator op,addrmode rm,  type t, offset sm)
        means[  (ref t) rm:= op(^(rm),(t) sm)  ]
assembles[op '  ' t ' ' rm ',' sm];
instruction  pattern INC(addrmode rm,int t)
means[(ref t)rm:=   + (^(rm),1)]
assembles['inc ' t ' ' rm];
instruction  pattern DEC(addrmode rm,int t)
means[(ref t)rm:=   - ((t)^(rm),1)]
assembles['dec ' t ' ' rm];
instruction  pattern SHIFT(shiftop op, shiftcount s, anyreg r,type t)
means[(ref t) r:= (t)op(^(r),s)]
assembles['sh' op' ' r ', 's];
instruction  pattern RMR( nonmultoperator op,addrmode rm,anyreg r1,wordupto32 t)
        means[ (ref t) rm :=op((t) ^( rm),(t)^( r1))]
     assembles[op '  '   t ' ' rm ',' r1];
instruction  pattern RMRB( nonmultoperator op,addrmode rm,breg r1,byte t)
        means[ (ref t) rm :=op((t) ^( rm),(t)^( r1))]
```

```
     assembles[op '   '   t ' ' rm ',' r1];
instruction pattern ADDUSB(addrmode fm,breg r1,breg rm)
means[ rm:= +:((uint8)^(rm),^(r1))]
assembles[ 'add  ' rm ',' r1 '\n jnc $+4\n mov ' rm',255\n nop\n nop'];
instruction pattern SUBUSB(breg r1,breg rm)
means[ rm:= -:((uint8)^(rm),^(r1))]
assembles[ 'sub  ' rm ',' r1 '\n jnc $+4\n mov ' rm',0\n nop\n nop'];
instruction pattern ADDSSB(breg r1,breg rm)
means[ rm:=(int8) +:((int8)^(rm),^(r1))]
assembles[ 'add  ' rm ',' r1 '\n jno $+10\n jg $+6\n mov 'rm' ,-128 \n jng $+4\n mov ' rm',127\n '];
instruction pattern MULTSSB(breg r1,bnonacc r2)
means[r2:=*:(^(r2),^(r1))]
assembles['push ax\n mov al,'r1'\n imul  'r2'\n shr ax,7\n mov 'r2',al\n pop ax'];

instruction pattern MULTSSBAL(bacc r1,bnonacc r2)
means[r1:=*:(^(r1),^(r2))]
assembles['imul  'r2'\n shr ax,7'];

instruction pattern SUBSSB(addrmode fm,breg r1,breg rm)
means[ rm:= (int8)-:((int8)^(rm),^(r1))]
assembles[ 'sub  ' rm ',' r1 '\n jno $+10\n jg $+6\n mov 'rm' ,-128 \n jng $+4\n mov ' rm',127\n nop\n nop'];
instruction pattern UINT8MAX(breg r1,breg r2)
        means[ (ref uint8)r1:=MAX((uint8)^(r1),^(r2))]
assembles['cmp 'r1','r2'\n ja $+4\n mov 'r1','r2];
instruction pattern UINT8MIN(breg r1,breg r2)
        means[ (ref uint8)r1:=MIN((uint8)^(r1),^(r2))]
assembles['cmp 'r1','r2'\n jna $+4\n mov 'r1','r2];
instruction pattern INT8MAX(breg r1,breg r2)
        means[ (ref int8)r1:=MAX((int8)^(r1),^(r2))]
assembles['cmp 'r1','r2'\n jg $+4\n mov 'r1','r2];
instruction pattern INT8MIN(breg r1,breg r2)
        means[ (ref int8)r1:=MIN((int8)^(r1),^(r2))]
assembles['cmp 'r1','r2'\n jl $+4\n mov 'r1','r2];
instruction pattern LEA(reg r1, eaform ea)
means [r1:=ea]
        assembles ['lea ' r1 ',[' ea ']' ];
instruction pattern NOTOP(addrmode rm, type t)
means[(ref t)rm:= NOT((t)^(rm))]
assembles['not 't ' ' rm];
instruction pattern Negate(anyreg r1,type t)
means[(ref t)r1:= -((t)0,( t)^(r1))]
assembles ['neg ' ' '  r1];
instruction pattern MNegate(anyreg r1,type t)
means[(ref t)r1:= *((t)-1,( t)^(r1))]
assembles ['neg '  ' '  r1];
instruction  pattern RLIT(operator op,pushreg r0,  type t, signed sm)
        means[r0:=  op(^( r0), const sm)  ]
assembles[op '  '  r0 ',' sm];
instruction pattern RR( nonmultoperator op, anyreg r1, anyreg r2, int t)
        means[r1:=(t) op((t) ^( (ref t) r1),(t)^( (ref t) r2))]
assembles[op ' ' r1 ',' r2];
instruction pattern iRR( operator op, reg r1, reg r2, int t)
        means[r1:=(t) op((t) ^( (ref t) r1),(t)^( (ref t) r2))]
assembles[op ' ' r1 ',' r2];
instruction  pattern RRM(operator op, pushreg r1, maddrmode rm, int t)
        means [r1:=(t) op((t) ^(r1),(t)^( rm))]
        assembles[op ' ' r1 ',' rm ] ;
pattern bnonacreg means[DH|DL|BH|BL|CH|CL];
```

```
pattern baccreg means[AL];
pattern baccregmode means[maddrmode|baccreg];
pattern bnonacregmode means[maddrmode|bnonacreg];
instruction pattern fastBIDIV(baccreg r1,bnonacregmode r2)
        means[r1:=div((int8)^(r1),(int8)^(r2))]
assembles[' movsx ax,'r1'\n idiv BYTE 'r2];
instruction pattern BIDIV(baccreg r1, bnonacregmode r2,baccregmode r3)
        means[r3:=div((int8)^(r1),(int8)^(r2))]
assembles[' movsx ax,'r1'\n idiv BYTE 'r2'\n mov BYTE 'r3',al'];
instruction pattern BIMUL(baccreg r1, bnonacreg r2)
        means[r2:=*((int8)^(r1),(int8)^(r2))]
assembles['imul BYTE 'r2'\n mov BYTE 'r2',al'];
instruction pattern fastIMUL(acc a,dacc d)
means[(ref int32)a:=*((int32)^(a),^(d))]
assembles['imul edx'];
instruction pattern IDIV(acc r1,modreg r2)
    means[PUSH(mainSTACK,div((int32)^(r1),^( r2))) ]
assembles['push edx\n cdq\n idiv   ' r2 '\n xchg eax,DWORD[esp]\n xchg eax,edx'];
instruction pattern UDIV(acc r1,modreg r2)
    means[PUSH(mainSTACK,div((uint32)^(r1),^( r2))) ]
assembles['push edx\n xor edx,edx\n div   ' r2 '\n xchg eax,DWORD[esp]\n xchg eax,edx'];
instruction pattern IMULLIT(pushreg r1,addrmode rm, signed s)
means[(ref int32)r1:=*(^(rm),const s)]
assembles['imul 'r1',DWORD 'rm','s];
instruction pattern IMOD(acc r1, modreg r2)
        means[PUSH(mainSTACK,MOD((int32)^(r1),^( r2))) ]
assembles['push edx\n cdq\n idiv   ' r2 '\n xchg edx,DWORD[esp]'];
instruction pattern UMOD(acc r1, modreg r2)
        means[PUSH(mainSTACK,MOD((uint32)^(r1),^( r2))) ]
assembles['push edx\n xor edx,edx\n div   ' r2 '\n xchg edx,DWORD[esp]'];
instruction pattern BIMOD(baccreg r1, bnonacreg r2)
        means[r2:=MOD((int8)^(r1),(int8)^(r2))]
assembles[' movsx ax,'r1'\n idiv 'r2'\n mov 'r2',ah'];

instruction pattern MOD2(reg r)
means[r:=MOD(^(r),2)]
assembles['and 'r ',1'];
instruction pattern MOD4(reg r)
means[r:=MOD(^(r),4)]
assembles['and 'r ',3'];
instruction pattern MOD8(reg r)
means[r:=MOD(^(r),8)]
assembles['and 'r ',7'];
instruction pattern DIV8(ureg r)
means[r:=div((uint32)^(r),8)]
assembles['shr 'r ',3'];

instruction pattern MOD16(reg r)
means[r:=MOD(^(r),16)]
assembles['and 'r ',15'];
instruction pattern PLANT(label l)
means[l]
assembles[l ':'];
instruction pattern PLANTRCONST( double r,type t)
means[loc (t)r]
assembles[ 'dq ' r];
instruction pattern PLANTICONST( longint r,type t)
means[loc (t) r]
```

```
assembles[ 'dd ' r];

instruction pattern PLANTSCONST( float  r,type t)
means[loc (t) r]
assembles[ 'dd ' r];
instruction pattern PLANTBCONST( byte r,type t)
means[loc (t) r]
assembles[ 'db ' r];
instruction pattern PLANTWCONST( word16 r,type t)
means[loc (t) r]
assembles[ 'dw ' r];
/*
```

Control transfers and tests

```
*/
instruction pattern FAIL(int i)
means[interrupt i]
assembles['int 'i];
instruction pattern GOTO(jumpmode l)
means[goto l]
assembles['jmp ' l];
instruction pattern IFLITGOTO(label l,addrmode r1,signed r2,condition c,signed t,int b)
means[if((b)c((t) ^(r1),const r2))goto l]
assembles[' cmp 't' ' r1 ',  '  r2 '\n j' c ' near  '  l];
instruction pattern BIFLITGOTO(label l,baddrmode r1,signed arg2,condition c,signed t)
means[if(c((t) ^(r1),const arg2))goto l]
assembles[' cmp 't' ' r1 ',  't  arg2 '\n j' c ' near  '  l];
instruction pattern IFGOTOB(label l,bireg r1,regaddrimmediate r2,condition c,signed t,int b)
means[if((int8)c( ^(r1),(int8) r2))goto l]
assembles['cmp ' r1 ',byte' '  '  r2 '\n j' c ' near  ' l];
instruction pattern IFGOTOW(label l,wreg r1,regaddrimmediate r2,condition c,signed t,int b)
means[if((int8)c( ^(r1),(int16) r2))goto l]
assembles['cmp ' r1 ',word' '  '  r2 '\n j' c ' near  ' l];
instruction pattern IFGOTO(label l,ireg r1,regaddrimmediate r2,condition c,signed t,int b)
means[if((int8)c( ^(r1),(int32) r2))goto l]
assembles['cmp ' r1 ',dword' '  '  r2 '\n j' c ' near  ' l];
instruction pattern IFASSp6(signedreg r1,regaddrimmediate r2,condition c,type t2,maddrmode r3,maddrmode rm, type t )
means[if((t2)c( (t)^(r1),(t) r2))(ref t)rm:= (t)^(r3)]
assembles['cmp ' r1 ','t '  '  r2
'\n mov 'r1','t rm
'\n cmov'c' 'r1','r3'\n mov 't rm ',' r1];
instruction pattern IFASS(signedreg r1,acc r2,maddrmode rm, type t,equals c,type t2 )
means[if((t2)c( (t)^(rm),(t) r2))(ref t)rm:= (t)^(r1)]
assembles['cmpxchg ' t rm ',' '  '  r1 ];

instruction pattern ANDIFGOTO(label l,anyreg r1,regaddrimmediate r2,condition c,int t,int b,anyreg r3,
                              regaddrimmediate r4, condition c2, int t2,int b)
means[if((b)AND((b)c((t) ^(r1),(t) r2),(b)c2((t2)^(r3),(t2)r4)))goto l]
assembles['andifgoto ' c','r1','t r2','c2','r3',' t2 r4 ',' l];
instruction pattern SET(condition c,signedreg r1,regaddrimmediate rm, breg r,signed t)
means[r:= c((t)^(r1),(t) rm)]
assembles['cmp 'r1 ','t ' ' rm '\n set'c ' ' r'\n sub 'r',1\n not 'r];
instruction pattern SETU(unsignedcondition c,unsignedreg r1,regaddrimmediate rm, breg r,unsigned t)
means[r:= c((t)^(r1),(t) rm)]
assembles['cmp 'r1 ','t ' ' rm '\n set'c ' ' r'\n sub 'r',1\n not 'r];
instruction pattern IFBOOL(label l,breg r1)
means[ if( (int8)^(r1))goto l]
assembles['test ' r1 ',' r1 '\n jnz near ' l];
```

```
instruction pattern BOUND0(reg r1,reg r2)
means[if(OR(<( ^(r2), ^((ref int32)mem(  ^(r1) ))), >( ^(r2), ^((ref int32)mem(+(^(r1), 4))))) )interrupt 5]
  assembles['bound ' r2 ',[' r1 ']'];
instruction pattern BOUND4(reg r1,reg r2)
means[if(OR(<( ^(r2),^((ref int32)mem(+(^(r1),4)))),>(^(r2),^((ref int32)mem(+(^(r1), 8))))))interrupt 5]
  assembles['bound ' r2 ',[' r1 '+4]'];
instruction pattern BOUND16(reg r1,reg r2)
means[if(OR(<( ^(r2),^((ref int32)mem(+(^(r1),16)))),>(^(r2),^((ref int32)mem(+(^(r1), 20))))))interrupt 5]
  assembles['bound ' r2 ',[' r1 '+16]'];
instruction pattern IFIN(reg r1,reg r2, label l)
means[ if((int8)AND((uint8)^(mem(r1 )) , <<( (uint8)1,^(r2))))goto l]
assembles['bt  [' r1 '],' r2 '\n jc 'l];

instruction pattern TESTIN(reg r1,reg r2, breg r)
means[ r:=AND((uint8)^(mem(r1 )) , <<( (uint8)1,^(r2)))]
assembles['bt  [' r1 '],' r2 '\n setc 'r'\n not 'r'\n inc 'r];
instruction pattern BTS(reg r1,reg r2)
means[(ref uint8)mem(r1 ):=OR((uint8)^(mem(r1 ) ), <<( (uint8)1,^(r2)))]
assembles['bts [' r1 '],' r2];
instruction pattern REPMOVSD(countreg s,maddrmode m1,sourcereg si, destreg di)
means[for (ref int32)m1:=0 to ^(s) step 1 do
              (ref int32)mem(+(^(di),*(^((ref int32)m1),4))):=^((ref int32)mem(+(^(si),*(^((ref int32)m1),4))))
]
assembles[' inc ecx\n rep movsd'];
instruction pattern REPMOVSB(countreg s,maddrmode m1,sourcereg si, destreg di)
means[for (ref int32)m1:=0 to ^(s) step 1 do
              (ref octet)mem(+(^(di),^((ref int32)m1))):=^((ref octet)mem(+(^(si),^((ref int32)m1))))
]
assembles[' inc ecx\n rep movsb'];



define(IA32codes,LOAD|MOVZXB|MOVSXB|MOVZXW|MOVSXW|MOVZXB2|MOVZXBW|MOVSXBW|
CLEARREG|STORELIT|INC|LEA|
RMLIT|RMR|
/* STOREWR|STORER|STOREBR|/* stores */

DEC|IMULLIT|
BIMUL|RLIT|RRM|RMRB|iRR|RR|DIV8|IDIV|fastBIDIV|BIDIV|UDIV
|MOD2|MOD4|MOD8|MOD16| IMOD|UMOD|
fastIMUL|
Negate|NOTOP|MNegate|BTS|TESTIN|SHIFT|
UINT8MAX|UINT8MIN|INT8MAX|INT8MIN|SELECT|
PLANT|LITPUSH| MEMPUSH|
SET|SETU|ToBYTE|IFASS|
ANDIFGOTO |IFLITGOTO|BIFLITGOTO|IFIN|IFGOTO|
IFGOTOB|IFGOTOW
|GOTO|FAIL|BOUND4|BOUND0|BOUND16
|PLANTBCONST|PLANTWCONST|PLANTICONST|PLANTRCONST|PLANTSCONST
|REPMOVSB|REPMOVSD |ADDUSB|SUBUSB|ADDSSB|SUBSSB|MULTSSB|MULTSSBAL )
/* these come last as they are a fallback for having no free registers */
define(lastIA32codes,RPUSH| REFPUSH|SDEREF|SDEREFDOUBLEWORD/* pushes */
|IFBOOL|SMLIT|SMRP|SADD|SMULIT|SMRADD|SOP|SMR|BSMR|/* stack ops */
                  STOREWR|STORER|STOREBR|/* stores */
STACKLOAD |REFPOP|MEMPOP|BPOP|BSPOP|BSPUSH|DMEMPUSH|RPUSHE
                         |BPUSH|STACKSTORE|STACKWSTORE|STACKBSTORE|RPOP|SMRSHIFT)
```

```
/*
```

*/

7.2 The MMX instruction-set

/*

7.2.1 MMX registers and instructions

Registers

```
*/

register doubleword MM0 assembles[ 'MM0'];
register doubleword MM1 assembles[ 'MM1'];
register doubleword MM2 assembles[ 'MM2'];
register doubleword MM3 assembles[ 'MM3'];
register doubleword MM4 assembles[ 'MM4'];
/* reserve for working space */
reserved register doubleword MM7 assembles[ 'MM7'];
reserved register doubleword MM5 assembles[ 'MM5'];
reserved register doubleword MM6 assembles[ 'MM6'];
/** used for operations using half registers */
alias register word MM0L=MM0(0:31) assembles['MM0'];
alias register word MM1L=MM1(0:31) assembles['MM1'];
alias register word MM2L=MM2(0:31) assembles['MM2'];
alias register word MM3L=MM3(0:31) assembles['MM3'];
alias register word MM4L=MM4(0:31) assembles['MM4'];
alias register word MM5L=MM5(0:31) assembles['MM5'];
/* used for 16 bit parallelism */
alias register int16 vector (4) MM016=MM0(0:63) assembles['MM0'];
alias register int16 vector (4) MM116=MM1(0:63) assembles['MM1'];
alias register int16 vector (4) MM216=MM2(0:63) assembles['MM2'];
alias register int16 vector (4) MM316=MM3(0:63) assembles['MM3'];
alias register int16 vector (4) MM416=MM4(0:63) assembles['MM4'];
alias register int16 vector (4) MM516=MM5(0:63) assembles['MM5'];

alias register int32 vector (2) MM032=MM0(0:63) assembles['MM0'];
alias register int32 vector (2) MM132=MM1(0:63) assembles['MM1'];
alias register int32 vector (2) MM232=MM2(0:63) assembles['MM2'];
alias register int32 vector (2) MM332=MM3(0:63) assembles['MM3'];
alias register int32 vector (2) MM432=MM4(0:63) assembles['MM4'];
alias register int32 vector (2) MM532=MM5(0:63) assembles['MM5'];

alias register int8 vector (8) MM08=MM0(0:63) assembles['MM0'];
alias register int8 vector (8) MM18=MM1(0:63) assembles['MM1'];
alias register int8 vector (8) MM28=MM2(0:63) assembles['MM2'];
alias register int8 vector (8) MM38=MM3(0:63) assembles['MM3'];
alias register int8 vector (8) MM48=MM4(0:63) assembles['MM4'];
alias register int8 vector (8) MM58=MM5(0:63) assembles['MM5'];

pattern im8reg means[MM48|MM38|MM58|MM08|MM18|MM28];
pattern im2reg means[MM432|MM332|MM532|MM032|MM132|MM232];
pattern im4reg means[MM416|MM316|MM516|MM016|MM116|MM216];
pattern untypedmreg means [MM1|MM3|MM4|MM5|MM2|MM0|MM7|MM6];
pattern wmreg means[MM1L|MM3L|MM4L|MM2L|MM0L|MM5L];
pattern lmreg means [MM1L|MM3L|MM4L| MM2L|MM0L| MM5L];
```

```
pattern mreg means[im4reg|untypedmreg|im8reg|im2reg];


/* define m4 macros to generate casts to the desired types */
define(shortquad, (int16 vector(4))$1)
define(refshortquad,(ref int16 vector(4))$1)
define(octoct,(int8 vector(8))$1)
define(octb,(octet vector(8))$1)
define(refoctb,(ref octet vector(8))$1)
define(octuint,(uint8 vector(8))$1)
define(refoctuint,(ref uint8 vector(8))$1)
define(refoctoct,(ref int8 vector(8))$1)
define(intpair, (int32 vector(2))$1)
define(refintpair,(ref int32 vector(2))$1)
/*
```

Addressing modes

```
*/
pattern mrmaddrmode means[maddrmode|mreg];

/*
```

MMX instructions

```
*/
instruction pattern PMULLW(im4reg m, mrmaddrmode ma)
means[m := shortquad(*(shortquad(^(m)),shortquad(^(ma))))]
assembles['pmullw ' m ',' ma];
instruction pattern PMULLSSB(im8reg m1,mrmaddrmode m2, mrmaddrmode ma)
means[m1:= octoct(*:(octoct(^(m2)),octoct(^(ma))))]
assembles['pxor MM7,MM7'   /* clear regs mm5 and mm7*/
  '\n pxor MM5,MM5'
  '\n punpckhbw MM7,'ma /* mm7 now has 4 words with the top 4 bytes of ma in them */
  '\n pxor MM6,MM6'
  '\n punpckhbw MM6,'m2
  '\n pxor 'm1','m1
                  '\n punpcklbw 'm1','ma
                  '\n pmulhw MM7,MM6'
                  '\n psraw MM7,7'
                  '\n punpcklbw MM6,'m2
                  '\n pmulhw 'm1',MM6'
  '\n psraw 'm1',7'
  '\n packsswb 'm1',MM7'];
instruction pattern MMXPUSH(mreg m)
means[PUSH(mainSTACK,m)]
assembles['sub esp,8\n movq [esp],'m];
instruction pattern MMXPOP(mreg m )
means[m:=POP(mainSTACK)]
assembles['movq 'm',[esp]\n add esp,8'];
instruction pattern PADDD(mreg m, mrmaddrmode ma)
means[refintpair(m) := intpair(+(intpair(^(m)),intpair(^(ma))))]
assembles ['paddd 'm ',' ma];
instruction pattern PADDW(im4reg m, mrmaddrmode ma)
means[refshortquad(m) := shortquad(+(shortquad(^(m)),shortquad(^(ma))))]
assembles ['paddw 'm ',' ma];
instruction pattern PADDB(im8reg m, mrmaddrmode ma)
means[refoctoct(m) := octoct(+(octoct(^(m)),octoct(^(ma))))]
assembles ['paddb 'm ',' ma];
```

```
instruction pattern CMPEQB(mreg m,mrmaddrmode ma)
means[refoctb(m):= octb(=(octb(^(m)),octb(^(ma))))]
assembles['pcmpeqb 'm','ma];
instruction pattern CMPGTB(mreg m,mrmaddrmode ma)
means[refoctb(m):= octb(>(octb(^(m)),octb(^(ma))))]
assembles['pcmpgtb 'm','ma];
instruction pattern CMPGTBR(mreg m,mrmaddrmode ma)
means[refoctb(m):= octb(<(octb(^(ma)),octb(^(m))))]
assembles['pcmpgtb 'm','ma];

 instruction pattern PADDUB(mreg m, mrmaddrmode ma)
means[refoctuint(m) := octuint(+(octuint(^(m)),octuint(^(ma))))]
assembles ['paddb 'm ',' ma];
instruction pattern PADDSB(im8reg m, mrmaddrmode ma)
means[m := octoct(+:(^(m),octoct(^(ma))))]
assembles ['paddsb 'm ',' ma];
instruction pattern PADDSB3(im8reg m,im8reg m2, mrmaddrmode ma)
means[m := octoct(+:(^(m2),octoct(^(ma))))]
assembles ['movq 'm','m2'\n paddsb 'm ',' ma];

instruction pattern PADDUSB(mreg m, mrmaddrmode ma)
means[refoctuint(m) := octuint(+:(octuint(^(m)),octuint(^(ma))))]
assembles ['paddusb 'm ',' ma];
/* stack add instructions */
 instruction pattern SPADDUSB(mreg m )
means[refoctuint(m) := octuint(+:(octuint(POP(mainSTACK)),octuint(POP(mainSTACK))))]
assembles ['movq 'm',[esp]\n paddusb 'm ',[esp+8]\n add esp,16'];
instruction pattern SPADDUB(mreg m )
means[refoctuint(m) := octuint(+(octuint(POP(mainSTACK)),octuint(POP(mainSTACK))))]
assembles ['movq 'm',[esp]\n paddb 'm ',[esp+8]\n add esp,16' ];
instruction pattern SPADDSB(im8reg m )
means[m := octoct(+:(octoct(POP(mainSTACK)),octoct(POP(mainSTACK))))]
assembles ['movq 'm',[esp]\n paddsb 'm ',[esp+8]\n add esp,16'];


instruction pattern SPSUBD(mreg m )
means[refintpair(m) := intpair(-(intpair(^(m)),intpair(POP(mainSTACK))))]
assembles ['psubd 'm ',[esp]\n add esp,8' ];
instruction pattern PSUBW(im4reg m, mrmaddrmode ma)
means[refshortquad(m) := shortquad(-(shortquad(^(m)),shortquad(^(ma))))]
assembles ['psubw 'm ',' ma];
instruction pattern PSUBB(im8reg m, mrmaddrmode ma)
means[refoctoct(m) := octoct(-(octoct(^(m)),octoct(^(ma))))]
assembles ['psubb 'm ',' ma];
instruction pattern PSUBUB(mreg m, mrmaddrmode ma)
means[refoctuint(m) := octuint(-(octuint(^(m)),octuint(^(ma))))]
assembles ['psubb 'm ',' ma];
instruction pattern PSUBSB(im8reg m, mrmaddrmode ma)
means[refoctoct(m) := octoct(-:(octoct(^(m)),octoct(^(ma))))]
assembles ['psubsb 'm ',' ma];
instruction pattern PSUBUSB(mreg m, mrmaddrmode ma)
means[refoctuint(m) := octuint(-:(octuint(^(m)),octuint(^(ma))))]
assembles ['psubusb 'm ',' ma];
instruction pattern PAND(mreg m, mrmaddrmode ma)
means[m := AND(^(m),^(ma))]
assembles ['pand 'm ',' ma];
instruction pattern PANDN(mreg m, mrmaddrmode ma)
means[m := AND(^(ma),NOT(^(m)))]
```

```
assembles ['pandn 'm ',' ma];
instruction pattern POR(mreg m, mrmaddrmode ma)
means[m := OR(^(m),^(ma))]
assembles ['por 'm ',' ma];
instruction pattern MOVDS(waddrmode rm, wmreg m)
means[(ref word)rm:= ^(m)]
assembles['movd 'rm ','m];
instruction pattern MOVDL(waddrmode rm, wmreg m)
means[m := (word)^(rm)]
assembles['movd 'm ','rm];
instruction pattern MOVOCTUINTL(addrmode rm, mreg m)
means[m := octuint(^(rm))]
assembles['movq ' m ',' rm];
instruction pattern MOVQS(maddrmode rm, mreg m)
means[(ref doubleword)rm:= ^(m)]
assembles['movq 'rm ','m];
instruction pattern MOVOCTUINTS(maddrmode rm,mreg m)
means[(ref uint8 vector(8))rm:=^(m)]
assembles['movq 'rm','m];
instruction pattern MOVQL(maddrmode rm, mreg m)
means[m := (doubleword)^(rm)]
assembles['movq ' m ',' rm];
instruction pattern MOVQLR(im8reg rm, im8reg m)
means[m := ^(rm)]
assembles['movq ' m ',' rm];

instruction pattern REP2(mreg m, reg r)
means[m:=rep(^(r),2)]
assembles['push 'r'\n push 'r'\n movq 'm',[esp]\n add esp,8'];

instruction pattern REP4(mreg m, wreg r)
means[m:=rep(^(r),4)]
assembles['push 'r'\n push 'r'\n push 'r'\n push 'r'\n movq 'm',[esp]\n add esp,8'];

instruction pattern REP8(mreg m, breg r)
means[m:=rep(^(r),8)]
assembles['sub esp,2\n mov [esp], 'r'\n mov [esp+1], 'r'\n xchg [esp],ax\npush ax\npush ax\n push ax\nxchg [esp+6],ax\n movq 'm',[esp]\n add esp,8'];
instruction pattern PACKSSDW(reg r, mreg m)
means[(ref int16 vector(4))m:=^((ref int32 vector(4))mem(r))]
assembles['movq 'm',['r']\n packssdw 'm',['r'+8]'];

instruction pattern PACKSSWB(reg r, im8reg m)
means[(ref int8 vector(8))m:=^((ref int16 vector(8))mem(r))]
assembles['movq 'm',['r']\n packsswb 'm',['r'+8]'];

instruction pattern PACKUSWB(reg r, mreg m)
means[(ref uint8 vector(8))m:=^((ref int16 vector(8))mem(r))]
assembles['movq 'm',['r']\n packuswb 'm',['r'+8]'];



define(mmxcodes,PADDD|PADDW|PADDB|PADDSB|PADDSB3|PADDUSB|PADDUB|SPADDSB|SPADDUSB|SPADDUB|PACKSSDW|PACKSSWB|PACKUSWB|
CMPEQB|CMPGTB|CMPGTBR|
SPSUBD|PSUBW|PSUBB|PSUBSB|PSUBUSB|PSUBUB|POR|PAND|PANDN|PMULLSSB|
               PMULLW|MOVDS|MOVDL|MOVQS|MOVQL|MOVQLR|MOVOCTUINTS|MOVOCTUINTL|REP2|REP4|REP8|
MMXPUSH)
```

```
define(athloncodes,PF2IW)
/*
```

*/

7.3 The 486 CPU

include(‘cpus/i386base.m4’) include(‘cpus/ifpu.m4’) /*

```
*/

instructionset [IA32codes|FSET|lastIA32codes|fpucodes|fpupushes]

/*
```

*/
/*

7.4 Pentium

/ include(‘cpus/i386base.m4’) include(‘cpus/ifpu.m4’) include(‘cpus/mmx.m4’) /

```
*/
instructionset [ IA32codes |lastIA32codes|fpucodes|fpupushes|mmxcodes ]

/*
```

*/

7.4.1 Concrete representation

Part II

VIPER

Ken Renfrew

Chapter 8

Introduction to VIPER

8.1 Rationale

When originally developed, Vector Pascal used a command line compiler operating in the classical Unix fashion. This interface is documented in section 5.1. However it has been conventional, at least since the release of UCSD Pascal in the late '70s for Pascal Compilers to be provided with an integrated development environment. The Vector Pascal IDE, provides the usual capabilities of such environments, but with the additional feature of literate programming support.

8.1.1 The Literate Programming Tool.

Today's pace of technological development seems to be rising beyond anything that could be conceived only a few decades ago. It is a common "joke" that any piece of modern technology is six months out of date by the time it reaches the show room.

Software development is one of the fastest moving areas of this technological stampede. With development happening at such a rate documentation is often at best a few steps behind the reality of the code of any system. Thus anyone attempting to maintain a system is left to their own ingenuity and some out of date documentation.

The constant updating of this documentation would in fact almost certainly be a more time consuming task than developing the program in the first place and hence time spent in this area can often be regarded as non productive time.

Several attempts have been made at automating this process. The automation process is often termed literate programming. The two most successful of these being `web` [23] a development of the TEX system which is the forefather of LATEX [25] developed by Leslie Lamport that is so widely used today, and JAVADOC. The JAVADOC system was developed by Sun Microsystems to document programs written in JAVA by including the document details inside specially marked comments [Sch1].

The Vector Pascal literate programming tool will combine these two approaches by allowing the programmer to embed LATEX commands with in special comment markers. These will still be able to be parsed by a conventional Pascal Compiler allowing the system to be used for conventional Pascal programming.

The embedding of LATEX commands in the program is not compulsory for those wishing to use the tool. There is a user selectable scale of detail that will be included automatically in documentation even from a normal Pascal program.

In addition in an attempt to make the programs idiosyncrasies more readable and to present the programs arguments more conventionally there is the option of using a "mathematical syntax converter" which will change some of the more impenetrable code into conventional mathematical symbolism [1]. The finished document being written, by the system in LaTeX to allow straight compilation into a postscript or pdf document formats.

To further aid the documentation the variables declared with in the program will be cross referenced to their instantiation point allowing a reader to cross reference a variable and thus remind themselves of it's exact nature.

This brief description clearly show the aids that a literate programming tool would bring to the programmer allowing documentation to be both kept up to date and in fact created retrospectively from existing code.

8.1.2 The Mathematical Syntax Converter.

A computer program by it's very nature has a structure which allows it to be read by a machine. Modern high level languages have abstracted themselves from this very successfully but never the less due to this underlying requirement the syntax of a program language can hide the program's algorithm from a human reader.

Programmers often use psuedo-code to explain algorithmic arguments. Mathematical notation is usually the most clear and precise way of presenting this argument. The mathematical converter allows a developer to use this system to convert the Pascal syntax into something closer to mathematical notation [2] and much more presentable to the human reader.

This feature is unique [3] in a programming interface and provides a further level of documentation. The documentation of the algorithms involved in the program, which are arguably the program's most valuable assets.

8.2 A System Overview

As can be seen from the rationale above the system breaks into three main sections. The program editor with the compiler, the literate programming tool and the mathematical syntax converter.

It is hoped that an improvement in performance of the supplied compiler can be achieved by statically loading the compilers class files for all target processors [4] at start up rather than the dynamic loading currently employed.

The I.D.E. will follow the traditional approach offering similar facilities to that of many other editors for different languages on the market place.

Among these facilities are a syntax highlighting (for Vector Pascal, LaTeX and HTML), a project manager with automatic make file facility, the ability to run a program in the environment with redirected input and output, a function & procedure finder linked to the source code, a error line highlighter for compilation errors, an external process runner for LaTeX compilers, TeX to HTML converters, a mini browser to show approximate results of the Literate programming tool etc...

The Literate programming tool has been described in it's rationale and incorporates the unique mathematical syntax conversion allowing a program to be converted to a mathematical argument at literally the touch of a button.

8.3 Which VIPER to download?

VIPER is platform independent for the operating systems it supports. These operating systems are: -

[1] Refer to separate section of for the rationale of the maths syntax converter.

[2] Precise mathematical notation although perhaps desirable is a more complex operation than the time allotted to the project would allow but none the less an interesting development for the future.

[3] Unique to the best of our knowledge at the time of submission.

[4] Processors currently supported are the Intel 486,Pentuin ,P3 and The Athalon K6.

- Linux
- Windows 9x
- Windows NT/2000/XP

The only decision to make on the VIPER download is whether the source code is required. The source version although much larger contains the source code for the VIPER I.D.E. and the Vector Pascal Compiler and all files required for a developer to further develop or adapt any of the systems within VIPER. The class file download provides the required files to have an operational VIPER installation.

8.4 System dependencies

VIPER depends on several pieces of software all of which are freely available to download from various sources. The vital dependencies are: -

- Java 1.3 or newer.
- The NASM assembler.
- The gcc linker. Included in Linux installations, for Windows use the cygwin or DJGPP versions of the gcc linker.

For full functionality the following systems are also required: -

- A LaTeX installation. LaTeX usually comes with Linux installations. The total MiKTeX package is recommended for all Windows installations.
- A dvi viewer usually included with a LaTeX installation. The YAP viewer included with MiKTeX is particularly recommended.
- A TeX to `HTML` converter. TTH was used in the development of the system.

It is recommended that all the above programs are set-up as per their own installation instructions and the appropriate class path established to suit the host machines operating system.

8.5 Installing Files

Assuming the VIPER files have been downloaded to a suitable place on the host machine the actual installation can begin. The only decision that must be made is where to install VIPER. VIPER can be installed anywhere on the host machine provided that there are no spaces in the directory path of the target directory.

Once this decision has been made the .zip file should be unzipped using a proprietary zip tool (e.g. WinZip, zip magic etc.) to the source directory.

When the .zip file has been unzipped there will be a directory called VectorPascal in the target directory. VectorPascal is the home directory of the VIPER system.

VIPER may be launched by : -

- All installations. Open a shell / DOS window change to the VIPER home directory and type the command `java viper.Viper` taking care of the capital letter.
- Windows installations. The batch file viper.bat is included in the VIPER home directory; running this will start VIPER. A shortcut to this batch file should be placed on the host machines desktop for the easiest start-up.
- Linux installations. The shell script viper.sh is included in the VIPER home directory; running this will start VIPER.

8.6 Setting up the compiler

VIPER detects the operating system installed at start up and then moves a suitable run time library into the ../VectorPascal/ilcg/Pascal directory where it will be available for the compiler. This is done automatically each time that VIPER is started.

The compiler options will need to be set-up along with the personal set-up proffered for the installation (see Chapter 9). The file type for the linker will need set-up. These options are: -

- For Linux or Windows using the Cygwin gcc use "elf".
- For Windows using the DJGPP linker use "coff".

It is important to read through the user guide (see Chapter 9) to avoid learning the system the painful way!

Chapter 9

VIPER User Guide

9.1 Setting Up the System

VIPER automatically sets the compiler flags to suit the operating system on the host machine. For those who have used the Vector Pascal compiler with a command line interface this means that the -U flag is set for Windows 9x and Windows NT installations, and not set for Linux/UNIX installations, the -o flag is set to produce an exe file with the same name as the Pascal source file. The .asm file and .o files are similarly named. If these flags mean nothing then that is not a problem, either ignore the preceding information or see the Vector Pascal reference manual in the help files of the VIPER system.

VIPER cannot however detect the versions of the gcc linker installed, this is left for the user. The -f flag of the compiler tells the compiler the file format to be used. To set this go to Set-Up / Compiler Options / Options and click the -f button and enter the file format into the adjacent text field. The format should be :

- Linux Installations and Windows installations with Cygwin gcc linker format is `elf`
- Windows with DJGPP linker format is `coff`

Figure 9.1: File Format Entries in Compiler Options

The other options on the Compiler options window are: -

- Smart (Not Yet Implemented on the V.P. compiler) Serializes / deserializes the code tree for the processor. This allows the compiler to ‘learn’ how to quickly respond to a given code segment.
- S suppresses the assembly and linking of the program (an assembler file is still produced).
- V causes the compiler to produce a verbose output to MyProg.lst when compiling MyProg.pas.
- CPUtag This option is used in conjunction with the -cpu option. It prefixes the .exe file with the name of the cpu for which the compiler is set. when this option is used the .exe cannot be run in the I.D.E.

- -cpu This option allow the source file to be compiled to a range of processors. To produce an exe file for a range of processors the CPUtag should be set. This prevents the exe file being over written by the next compilation for a different processor. Subsequent compilations for the same processor, however, will be overwritten. Select the cpu from the list in the drop down menu adjacent to the -cpu button.
- -ISO (Not Yet Implemented on the V.P. compiler) Compiles to iso standard Pascal.

9.1.1 Setting System Dependencies

VIPER depends on various other systems for full functionality. These are set in Set-Up / Compiler Options / Dependencies The fields are: -

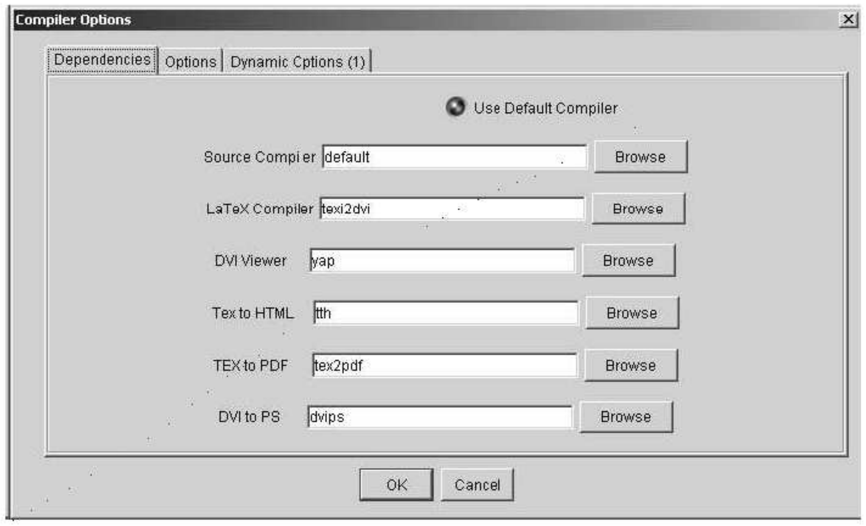

Figure 9.2: Dependencies Window

1. Source Compiler this option is only editable if the Default Compiler option is not set. This is the command that would run the compiler from the VectorPascal directory.
2. This is the command required to run LaTeX this is required for VPTeX to work. The recommended option for this field is `texi2dvi`.
3. DVI viewer The dvi viewer that is to be used to view the LaTeX recommended option is YAP (Windows installations).
4. Tex to HTML if a converter is installed on the host machine then put the command in this field.
5. Tex to PDF enter the command used to convert tex to PDF.
6. DVI to PS command to convert DVI files to PostScript (usually dvips).

9.1.2 Personal Set-up

Viper allows the user many options to cater for different tastes and programming styles. It is not crucial to the system to set these options but it does make for a more comfortable programming environment.

If your VIPER installation is on a network each user may have a different personal set-up providing each user has a separate home directory. VIPER installs a file called `viper.properties` into this directory and updates this file when ever a change is made to the system set-up.

NOTE The individual set-up should not be attempted when multiple files are open. If this is done then no harm comes to the system or any of the open files but users may experience difficulty in closing one or more files. The solution is to use Window / Close All to close all the files. The system can then be used as normal.

Viper Options

In the Set-up menu there is the Viper Options menu option. In this you will find all the familiar I.D.E. options such as font size and style, icons sizes, syntax colours, look and feel etc.

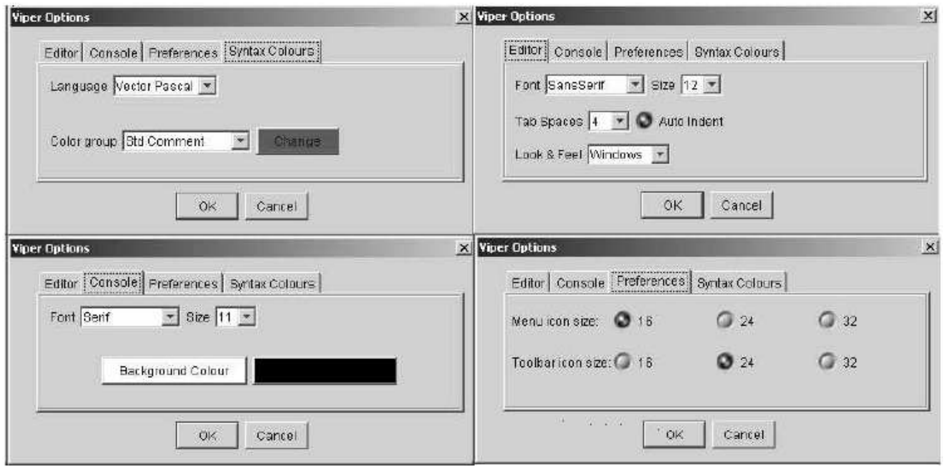

Figure 9.3: The Viper Option Windows

The different Windows shown above allow the control of the VIPER I.D.E. The individual windows control: -

- **Editor** This controls the look and feel (see Colour Plates) the font size and style, the tab size and auto indentation.
- **Console** This controls the Font style and size and the background colour of the console window.
- **Preferences** This allows the individual set-up of the menu icon sizes and the tool bar sizes.
- **Syntax Colours** This allows the Syntax Highlighting colours to be altered to personal taste. These can be adjusted for each supported language (Vector Pascal, LaTeX, HTML) independently.

9.1.3 Dynamic Compiler Options

NOTE This is for advanced use only.

This feature is intended to allow VIPER to handle: -

- New processors as the class files become available (Dynamic class loading only).
- New options for the compiler / new versions of the compiler.

The dynamically created options pages are added in the form of a new tabbed pane to the Compiler Options window. To create a new options pane the user must: -

1. Open the file ../VectorPascal/viper/resources/dynamicOption.properties
2. Edit the file to suit the new options.
3. Save the file.

Editing to add a processor

In the file dynamicoptions.properties in the ../VectorPascal/viper/resources directory there is a list of the current processors.. This list can be extended simply by adding another to the end of the list. It is best if the list ends with “others”.

Note The appropriate code generator files must be written for the Vector Pascal compiler and placed in the ../VectorPascal/ilcg/tree directory.

Editing to add compiler options

The dynamicoptions.properties file can be edited to produce a new compiler option. This is done by entering a new line at the end of the file following the in the line above. For example: -

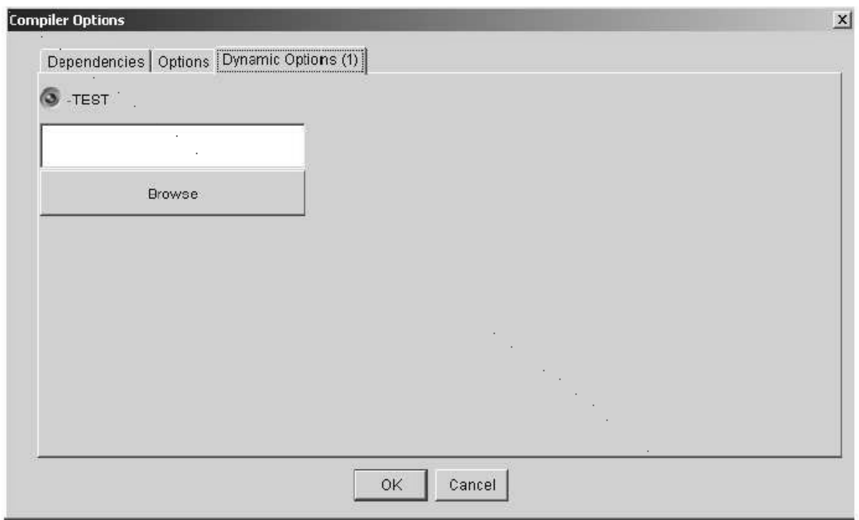

Figure 9.4: Dynamic Option Window

```
CPUFLAGS: P3:K6:Pentium:IA32
#
#This is to set flags for the compiler
#NB DO NOT EDIT THIS FILE BEFORE AFTER READING THE HELP FILE
#IT IS IMPORTANT THAT THE FIELDS COME IN THE FOLLOWING ORDER
#FLAG(Type:String),DESCRIPTION(Type:String),TEXTFIELD(Type:int),
#BROWSEBUTTON(Type: boolean)
#Any comments must be but in this area.

FLAG:DESCRIPTION :TEXTFIELD: BROWSEBUTTON
-TEST:Test description: 20 : true:
```

9.1.4 VIPER Option Buttons

The VIPER options are set in their respective panels with the VIPER Option Buttons these have three states: -

- **Grey** The item is not selected.
- **Red** The mouse is over the correct areas to select the item.
- **Blue** The item is selected.

9.2 Moving VIPER

Ideally VIPER should be installed from the downloaded zip file on any new system. If this is not possible then it is still possible to move VIPER onto a new system even if the new host machine has a different operating system.

Moving a VIPER installation from any Windows host to any other Windows host, or from one Linux installation to another is straight forward.

1. Move the entire VectorPascal directory and all sub-directories to the new system.
2. Run VIPER and in the File menu click clear recent files and then click clear recent projects.
3. Import all projects that have been moved and are to be used on the new system.

If the operating systems are different (i.e. moving from Linux to Windows or vice versa) then the system must be reset: -

1. open a shell/DOS prompt window and change directories to the VectorPascal directory.
2. Type `java ViperSystemReset` in the console window.

The system is now reset and the new installation of VIPER can be used normally.

9.3 Programming with VIPER

This section assumes that the I.D.E. is now set-up to the user's taste. To open a file click the open file menu option and use the dialogue box to open the file in the usual way.

Familiarity with the basic editing functions of an I.D.E. are assumed.

9.3.1 Single Files

The file will open with the syntax highlighter associated with the file suffix of the target file. The file can be edited with all the usual I.D.E. functions. (Cut, Paste, Copy, Save, Save As, Find and Replace, etc.).

VIPER features a "right click menu" to offer another method of quickly editing files.

Line numbers can be viewed either by using the statistics on the status bar at the bottom right hand corner of the I.D.E. or by double clicking the dark grey panel on the left of the editor window, this line number panel can then be adjusted in size to suit the user's needs.

A new file can be opened from the file menu. Clicking on the New Document option allows the user to choose between the three types of file that VIPER supports (.Pascal, LaTeX, HTML). A new file is then opened in the editor window. The file is un-named until it has been saved.

When a file has been changed since it was last saved the name tag at the top of the editor window appears in red, otherwise it is black.

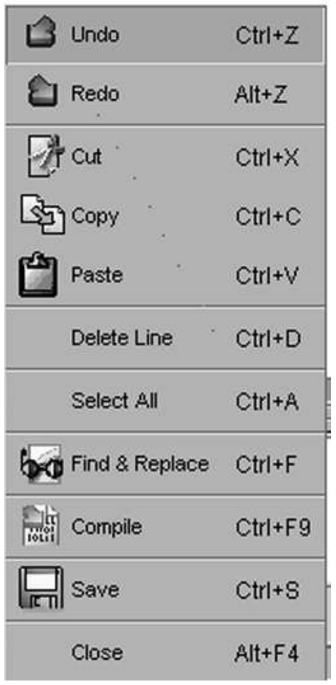

Figure 9.5: The Right Click Menu

If the user attempts to close the editor before a file is saved the option to save the file is offered before the I.D.E. closes.

If a file has functions and / or procedures the function finder automatically displays these in the left most editor window. Clicking on the icon by a function or procedure takes the editor to the start of that section.

9.3.2 Projects

The VIPER Project Manager allows the user to construct software projects in Vector Pascal.

An existing project can be opened using the Project / Open Project menu option or icon. The project will then appear in the project window. The files names are in a tree structure which can be clicked to open the file in the editor window.

To create a new project the user clicks on the new project icon and the Project properties dialogue box will appear.

The text fields are then filled in to create the empty project. The directory path should be the parent directory for the project's home directory. This home directory will be given the project's name.

Once the project has been created the files can be added and removed as required.

- **Adding** Click the add files icon and enter or browse for the required file. This copies the file to the project directory.

- **Removing** Highlight the file too be removed and click the remove files icon. **Warning** This deletes the file from the project directory.

Other files may be placed in the project directory but if they are not added to the project they will not be a member of the project.

The makefile for the project is automatically created as ProjectName.mke. The user should not edit either this or the .prj file directly.

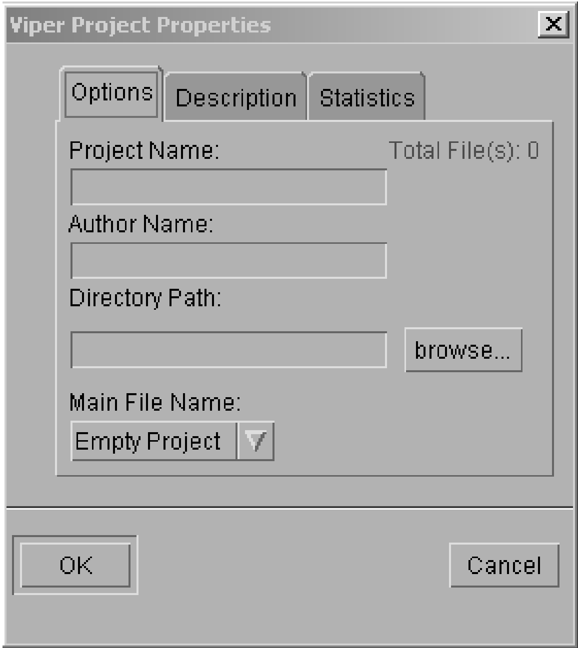

Figure 9.6: The Project Properties Window

Importing Projects

Projects can be imported from other VIPER installations by the import project facility. This can be found in Project / Import Project. Any project coming from another VIPER must be imported via this facility.

Backing-Up Projects

The import project facility can be used to move an existing project to another directory of the same machine. This Back-Up project is not just a copy of the project but is fully functional with all the facilities of the VIPER system.

9.3.3 Embedding LaTeX in Vector Pascal

The special comment (*! comment body *) is used to embed LaTeX in the Vector Pascal source file. Anything in within these comments will be treated as if it were LaTeX both by the VPTeX system and the syntax highlighter.

There is no need to put LaTeX commands in the special comments unless a specific result is required. (See section 9.9)

9.4 Compiling Files in VIPER

9.4.1 Compiling Single Files

Assuming the compiler has been set-up the compilation of a file is very simple. Simply click the compile icon (or menu option) and the compiler will compile the file in the editor window with the options selected.

The resulting files are placed in the same directory as the source file and are named the same as the source file with the corresponding suffix.

Compiling a file to executable for several processors

If a file is to be compiled for several different processors the CPUTAG and -CPU options must be set in the Set-Up / Compiler Options / Options panel. The file MyProg.pas would then be compiled to ProcessorNameMyProg.exe. This process can be done for each processor on the available processor list.

Note A file compiled in this manner cannot be run within the I.D.E.

9.4.2 Compiling Projects

Projects can be compiled in two ways: -

- Make a project. This compiles the files that are not up to date but does not compile any file that is up to date.
- Build a project. This compiles all the files in the project regardless of whether the files are up to date.

The Vector Pascal compiler used in the traditional command line interface mode will check one level of dependency in a project. If there are more levels of dependency the VIPER project manager will automatically make a `makefile` and recursively check all levels of dependency in the project.

As VIPER compiles a file, the file is opened in the I.D.E. if an error is found compilation stops and the error is highlighted.

9.5 Running Programs in VIPER

Note Projects requiring input from the user **MUST** have the input redirected.

When a program has been compiled the resulting executable can be run in the I.D.E. by clicking on the Run icon. A redirect input box then appears. If the program requires input from the user then an input file must be set. This file should contain all the data that the program requires to run to completion.

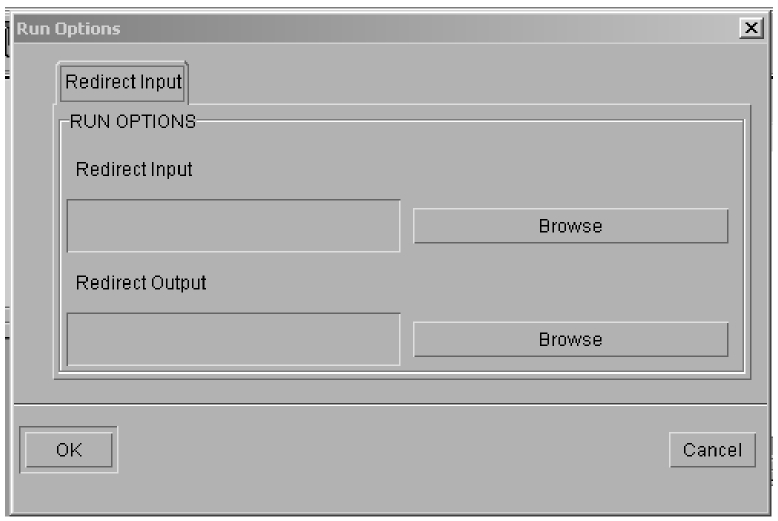

Figure 9.7: The Run Options Panel

Similarly the output may be redirected. This, however is not compulsory if the output is not redirected the output of the program appears in the console window. If the output is redirected then the output is written to the file set-up in the run dialogue window.

9.6 Making VPTEX

Making VPTEX is as simple as clicking the Build VPTEX icon or menu option. If a project is open then the VPTEX is made for the whole project, otherwise the VPTEX is made for the file in the editor window.

9.6.1 VPTEXOptions

The level of documentation is set by the user in the VPTEX Options panel. This panel can be found in the TeX / VP-TeX Options menu item. There are five levels of detail that can be chosen :-

- Function and Procedure headings only.
- Level 1 plus all special comments.
- Program bodies and interfaces.
- Selected text
- All source code.

In addition to the above options the user can choose whether a contents page is to be included or not. This is set by clicking the create contents page button.

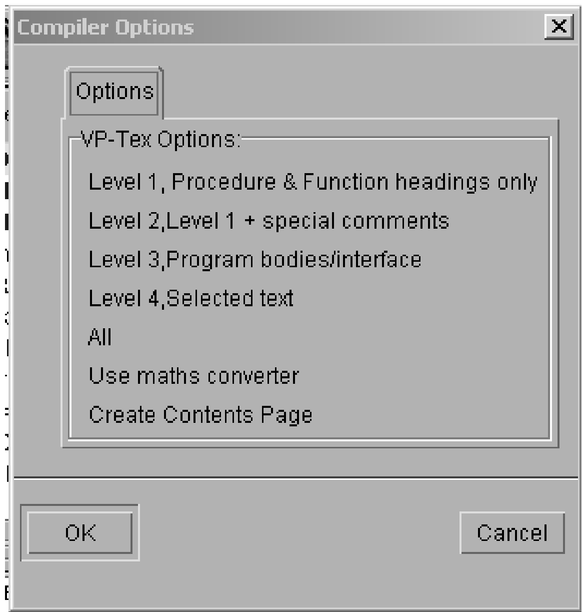

Figure 9.8: The VPTEX Options Panel

9.6.2 VPMath

The VPMath system converts Vector Pascal code to mathematical syntax. This makes the program more human readable and in general more concise.

The VPMath system is invoked automatically when the VPTeX is made if the Use Math Converter is set in the Tex/VP-TeX Options menu item.

9.7 LaTeX in VIPER

Most of the features of the VIPER editor used in the creation / editing of Vector Pascal files can also be used for creating / editing LaTeX documents.

Opening a LaTeX document in VIPER automatically invokes the LaTeX syntax highlighter and the Function Methods finder automatically changes to a Section / Sub-Section finder.

This allows the user to click on a Section icon in the left hand window and the editor will jump to that section.

9.8 HTML in VIPER

VIPER allows the user to edit/write HTML pages. The system for HTML is very straight forward. Create a new HTML file or open an existing file to be edited. Once the file has been altered click on the run button just as if to run a Vector Pascal executable.

When a new HTML file is created or an existing one opened the HTML syntax highlighter is automatically loaded.

The default browser that is installed on the host machine will open with the HTML page displayed.

9.9 Writing Code to Generate Good VPTeX

VPTeX is a tool included in the VIPER Integrated Development Environment for Vector Pascal. It automatically produces and formats a LaTeX listing of the source file or files on which it is called. By defining three distinct types of comments, VPTeX also allows the programmer to add extensive descriptions of their code to the listing, creating full LaTeX documentation for their Vector Pascal programs or projects. Mathematical translation can also be performed on the source code listing to produce a more generic and succinct description of the program's algorithms and structures.

The three types of comments available are:

Special Comments : A special comment is started in the source code with the comment command (*! and terminated with *). Special comments appear in the LaTeX as running prose and are of most use in giving extensive comments and descriptions of the program. Special comments can include LaTeX commands, with some limitations, to further imrove the readability of the documentation.

Margin Comments : Normal Pascal {...} comments which appear immediately at the end of a line of code are placed in the left-hand margin adjacent to their source code line in the LaTeX documentation. These are of principal use when a small description of the content of a single line is required.

Normal Comments : Normal Pascal {...} comments which appear on a line of their own will appear in the LaTeX in typewriter font.

9.9.1 Use of Special Comments

As outlined above, special comments are the principal means of describing a program in the documentation. To maximise the effectiveness of the literate programming facility source code should be written with large amounts of special comments and with the program's documentation in mind. The ability to include LaTeX command within special comments allows the programmer to directly affect the look of the LaTeX documentation, but there are some limits to the use of LaTeX commands within special comments:

- Do not include any preamble within special comments. The preamble for the LaTeX documents is automatically produced by VPTeX.
- Always use full text series altering commands such as `\textbf{..}` rather than their shorthand equivalents such as `\bf{...}`.
- Bear in mind that any text entered in special comments must be compilable LaTeX for the documentation to compile. This means that the following characters are control characters and should not be entered verbatim into special comments; & $ % _ { } ^ ~ \.

Special comments can be particularly useful for controlling the structure of your LaTeX document. The following are guidelines as to how to structure your documentation.

- For an individual program or unit file, the LaTeX document produced by VPTeX will be an article, so sections are the highest level description that can be applied to a block of text.
- It is usually useful to include an introduction to the program at the start of the Pascal source file using the `\section{Introduction}` comand at the start of an opening special comment.
- A special comment containing just a structure command (`\section, \subsection` etc.) can be extremely useful in sectioning off different parts of the source code to add structure to the code listing. For example, the declarations could be prefaced with `(*! \section{Declarations} *)` or the main program could be prefaced with a similar command. Each procedure or function is automatically placed within its own section by VPTeX so do not add structuring special comments to these sections of code.

To produce a well documented program, it is important that special comments are regularly employed to add verbose descriptions of the source code. It is not uncommon for a LaTeX documentation file to contain many pages of special comments split into sections and subsections between small sections of code. VPTeX also automatically creates a content page so the structure of your special comments will be reflected in the content page.

Note: With the current release of the Vector Pascal compiler, special comments containing *'s other than at the opening (*! and closing *) tags will not compile.

9.9.2 Use of Margin Comments

Margin comments are useful for providing short descriptions of the purpose of individual lines of code. If the meaning of a particular code line is especially cryptic, or the significance of the line needs to be emphasised, a margin comment stating the purpose of that line may be useful. Please be aware that because margin comments necessarily reside in the left-hand margin of the finished document, lengthy comments will spill onto many lines and break up the flow of the code. It is advised that margin comments should not be more than 10 or so words, with the other types of comments available if a longer description is required.

The VPTeX tool automatically breaks lines following the **var** and **const** keywords. Therefore, the declaration following these keywords will be placed on a new line, but any margin comment for this line will not. It is recommended that the programmer takes a new line after the **var** and **const** keywords.

9.9.3 Use of Ordinary Pascal Comments

The function of normal Pascal comments has been superceded in most cases by VPTeX's Special Comments. However, normal comments can still be useful in a number of circumstances. The following list details the recommended usage of normal Pascal comments, but the user is, of course, free to make use of them in any circumstances he wishes.

- Firstly, because normal comments are displayed in typewriter font, any spacing within these comments set out by the programmer will be preserved in the documentation. This is not the case for special comments which are displayed in a serifed, variable width font. This property of normal comments makes them particulary suitable for laying out tables and arrays simply, although a special comment can make use of LaTeX's ability to typeset tables for a more advanced layout.
- Secondly, normal comments do not break up the flow of a code listing to the same extent as special comments and so are more useful for offering a running commentary on code lines, without the space limitations of margin comments.
- If a comment is reasonably short, the programmer may find that a normal comment will have a better appearance than a special comment. Since special comment are offset from the program listing a small special comment may constitute a waste of the space set aside for it.

9.9.4 Levels of Detail within Documentation

Depending on the sort of documentation you want to produce, VPTeX allows the programmer to specify the detail of their program documentation. The five levels are:

1. **Procedure and Function Headings Only:** For documentation of ADT's it is often useful to simply provide a list of the functions and procedures by which a programmer may make use of the ADT. VPTeX supports this by providing the option to create documentation consisting of only function and procedure headings. It is advised that a contents page is not included with this level of detail.
2. **Special Comments with Function and Procedure Headings:** To add commentary and descriptions to the above level of detail, option 2 will add any special comments to the documentation. This allows the programmer to provide descriptions of their procedures and functions and to add structure to the documentation. A contents page is advised for this level of detail.
3. **Program Bodies and Unit Interfaces:** This level of detail includes all comments. It is again very useful for documenting ADT's as the interfaces provided by units will be documented, but none of the implementation will be included. A contents page is recommended.
4. **Selected Text:** Special VPTeX comments commands have been defined to allow the programmer to select which sections of their program to document. The commands are `(*!begin*)` to mark the start of a selected region, and `(*!end*)` to mark the end. Any text, including special comments, not contained within these tags will be ignored by VPTeX if this level of detail is selected. The start and end of the main program file will always be included in the documentation regardless of selection. This feature is of particular use when preparing reports regarding particular sections of code within long projects as only the sections of interest will be documented. Again, a contents page is recommended.
5. **All Code and Comments:** For a completely documented code listing, of particular use for system maintenance, VPTeX can produce a complete listing of a program or

project's source code, including special and normal comments. A contents page is strongly recommended, particularly for long programs or projects.

Note: All levels of detail support margin comments.

9.9.5 Mathematical Translation: Motivation and Guidelines

VPTeX has the option of automatically translating the program code into conventional mathematical notation. Complex VectorPascal expressions like:

myVariable:= if (iota 0 div 2 pow (dim-iota 1)) mod 2 = 0 then 1 else -1;

are translated into more tidy and comprehensible mathematical representations like.

$$\mathit{myVariable} \leftarrow \begin{cases} 1 & \text{if } (\frac{\iota_0}{2^{dim-\iota_1}}) \textbf{ mod } 2 = 0 \\ -1 & \text{otherwise} \end{cases} ;$$

No action is required to get mathematical translation, so long as it is turned on (VP-TeX Options), however the benefits of using it increse with the number of mathematical structures in the document. In particular, the following will benefit from mathematical translation:

- Array indexing/slicing, e.g thisArray$_{i,j}$ / thatArray$_{low..high}$
- Assignments, e.g. myVarable $\leftarrow$ yourVariable
- Reduction operations on arrays, e.g myVariable $\leftarrow$ ΣoneDArray
- Conditional updates (as shown above)
- A number of standard mathematical function such as square root
- Mathematical operations, e.g. x^y, $\frac{a}{b}$, $i \times j$
- English names of Greek letters (lower case only), e.g. α, β, γ, δ

Mathematical translation is particularly useful if the documentation is for people without knowledge of Pascal or a similar language. The only time mathematical translation is not advisable is when the reader is maintaining the code itself, in which case the need for cross reference will usually dominate the need for clarity and conventional notation.

9.9.6 LaTeX Packages

All VPTeX documents only include packages `graphicx` and `epsfig`. These packages are included to allow the programmer to include graphics and diagrams to help document their programs. Any LaTeX commands the programmer may wish to use which are specific to other packages cannot be included in VPTeX special comments.

Index

Bibliography

[1] 3L Limited, Parallel C V2.2, Software Product Description, 1995.

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

[3] 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.

[4] Blelloch, G. E., NESL: A Nested Data-Parallel Language, Carnegie Mellon University, CMU-CS-95-170, Sept 1995.

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

[6] 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.

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

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

[9] Cherry, G., W., Pascal Programming Structures, Reston Publishing, Reston, 1980.

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

[11] Ewing, A. K., Richardson, H., Simpson, A. D., Kulkarni, R., Writing Data Parallel Programs with High Performance Fortran, Edinburgh Parallel Computing Centre, Ver 1.3.1.

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

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

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

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

[16] ISO, Pascal, ISO 7185:1990, 1991.

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

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

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

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

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

[22] Johnston, D., C++ Parallel Systems, ECH: Engineering Computing Newsletter, No. 55, Daresbury Laboratory/Rutherford Appleton Laboratory, March 1995,pp 6-7.

[23] Knuth, D., Computers and Typesetting, Addison Wesley, 1994.

[24] Krall, A., and Lelait, S., Compilation Techniques for Multimedia Processors, International Journal of Parallel Programming, Vol. 28, No. 4, pp 347-361, 2000.

[25] Lamport, L., LaTeXa document preparation system, Addison Wesley, 1994.

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

[27] Marx, K., 1976,*Capital*, Volume I, Harmondsworth: Penguin/New Left Review.

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

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

[30] Shannon, C., A Mathematical Theory of Communication, The Bell System Technical Journal, Vol 27, pp 379-423 and 623-656, 1948.

[31] Snyder, L., A Programmer's Guide to ZPL, MIT Press, Cambridge, Mass, 1999.

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

[33] Strachey, C., Fundamental Concepts of Programming Languages, University of Oxford, 1967.

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

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

[36] Wirth, N., Recollections about the development of Pascal, in *History of Programming Languages-II*, ACM-Press, pp 97-111, 1996.