ZPL Language Reference Manual[*]

Calvin Lin

Department of Computer Science and Engr., FR-35
University of Washington
Seattle, WA 98195

Version 0.2
January 16, 1995

[*]This research was supported in part by ARPA Grant N00014-92-J-1824 and NSF Contract CDA-9211095

Contents

1 Introduction

This document presents a complete description of the ZPL language. The language is presented without motivation in a bottom-up fashion, starting with lexical elements and then building upon these to introduce the ZPL data types, variables, expressions, and statements. An extensive index is found at the end.

2 Lexical Issues

2.1 Comments

ZPL has two types of comments. Dashes (`--`) specify comments that continue to the end of the line (unless the dashes appear in a string literal). Multiple line comments are bracketed by `/*` and `*/` (unless these characters appear in a string literal).

2.2 White Space

All white space—blanks, tabs, newlines and linefeeds—are ignored except to separate tokens.

2.3 Case-Sensitivity

ZPL is case-sensitive.

2.4 Tokens

ZPL tokens fall into one of the following categories: identifiers, keywords, constants, operators, and separators. Each category is described below.

2.4.1 Identifiers

An identifier is a programmer-defined name that consists of a sequence of letters, digits or the underscore character (_) and begins with a letter. Identifiers can be used to name new data types, variables, record fields, named constants, procedures, and program names.

2.4.2 Keywords

Table 1 lists the reserved words in ZPL that cannot be used as identifiers.

2.4.3 Constants

Numeric constants are by default specified in decimal units. Octal and hexadecimal constants are specified using a leading `0` and `0x` (or `0X`), respectively. Thus, the following are equivalent: `15`, `017`, `0xf`. Constants can be specified to be a `longint` by using a trailing `l` or `L`. Otherwise, their type is `integer`.

Table 1: Keywords in ZPL.

and	array	begin	by	char	config
constant	continue	direction	double	do	downto
else	elsif	end	exit	file	for
halt	if	integer	longint	of	or
procedure	program	prototype	read	real	record
reflect	region	repeat	return	shortint	string
then	to	type	union	unsigned	until
var	while	with	without	wrap	write
writeln					

Floating point values can contain a decimal point (`.`) and an exponent (`e` or `E`). For example, `0.001` and `1e-3` are equivalent. By default, floating point constants are of type `real`; a `double` is specified by using a trailing `l` or `L`.

Character constants are delimited by single quotes. For example, the character `a` is specified by `'a'`. A list of special characters is shown in Table 2.

Table 2: Special Characters in ZPL.

`'\b'`	backspace
`'\f'`	formfeed
`'\h'`	hexadecimal escape sequence
`'\n'`	newline
`'\o'`	octal escape sequence
`'\r'`	carriage return
`'\t'`	tab
`'\v'`	vertical tab
`'\0'`	end of string
`'\\'`	backslash

Other ASCII characters can be specified numerically by using the octal and hexadecimal escape sequences to specify octal and hexadecimal bit sequences. For example, `'\x07'` specifies the ASCII character whose numeric value is 7 in hexadecimal (this happens to be the bell character).

String constants are delimited by double quotes ("). String constants may not span multiple lines, but consecutive string constants are automatically concatenated by the compiler to represent a single long string constant. Thus, the following are equivalent:

```
"hello world"
```

```
"hello " "world".
```

Named Constants. Constants can be given a name. These are defined at the global scope. See Section 8.1.

2.4.4 Operators

Table 5 lists the ZPL operators. These operators are described in Section 5.

2.4.5 Separators

The following symbols serve as token separators in ZPL.

```
:   ..   ,   ;   .
```

The use of colons (`:`) is described in Section 4.2. When declaring variables, the colon separates variable names from their types. The use of dots (`..`) and commas (`,`) is shown in Sections 3.3.3 and 3.5. These dots separate the lower and upper bounds of an array dimension, and the comma separates the items in a list of dimensions. The semicolon (`;`) is a statement terminator: All ZPL statements end with a semicolon. Finally, Section 3.3.1 describes how the period (`.`) is used to specify fields of a record variable.

3 Data Types

3.1 Base Types

Table 3 shows the base data types that are supported in ZPL. All of the base types except `file` and `string` are considered to be *arithmetic types*. The arithmetic types all have C equivalents, which is important when invoking external C procedures (see Section 8.2). The *integral* types consist of all arithmetic types except for `float` and `double`.

The `file` data type is used to specify file I/O (Section 7.4 describes file I/O). Variables of type `file` can be assigned to, passed to procedures as parameters, and returned from procedures; they cannot be used as operands to any of the operators listed in Table 5 (see page 20).

The `string` data type represents a sequence of characters. Variables of this type may be assigned to, initialized as config parameters (Section 4.1.1 describes config parameters), and passed to and returned from procedures.

3.2 Enumerated Types

An enumerated type is a user-defined set of integral values. Each element of an enumerated type is given a unique identifier (by the user) and a unique value. By default, the values of an enumerated type are consecutive starting at `0`. Thus, it is legal to iterate from one element to another, as shown below.

Table 3: Base Types in ZPL (and their C equivalents).

char	(char)
integer	(int)
shortint	(short)
longint	(long)
unsigned char	(unsigned char)
unsigned integer	(unsigned int)
unsigned shortint	(unsigned short)
unsigned longint	(unsigned long)
real	(float)
double	(double)
file	
string	(char *)

```
type fruit = (apple, banana, orange, mango);

var f : fruit;

for f := apple to orange do
    . . .
end;
```

All operations that are legal on integral types are legal on enumerated types, except that the value of an enumerated type is limited to the set of values specified in the enumerated type's declaration.

The values of specific elements of an enumerated type may be specified in the type declaration. The following declaration would start the values at `171` instead of `0`.

```
type fruit = (apple=171, banana, orange, mango);
```

Any arbitrary integral expression can be used to define values of enumerated types. These expressions cannot refer to other values in the same set, so the following is illegal.

```
type fruit = (apple=171, banana=apple+2, orange, mango);
```

3.3 Derived Types

New data types can be constructed using `records`, `parallel arrays`, `indexed arrays`, and `unions`.

Table 4: Derived Types in ZPL.

record
parallel array
indexed array
union

3.3.1 Records

Records provide a way to associate related data into a single data structure. A record consists of one or more *fields* that each have a name and a type. The type of a field may be any base or derived type. The names must be unique within each record type, but each record type defines a new lexical scope. A record can be declared as illustrated below:

```
type person = record
       age    : integer;
       weight : integer;
     end;
```

Individual fields of a record are accessed using the dot operator (`.`):

```
var  daphne : person;

daphne.age := 10;
```

Records can be assigned or passed as parameters to procedures as whole entities. When records are passed as value parameters, only the top level fields are copied.

3.3.2 Parallel Arrays

Parallel arrays are the primary means of specifying parallelism in ZPL. While the use of this data type has sequential semantics, the compiler will distribute these for execution on parallel machines. Parallel arrays are declared using regions (Regions are described in Section 3.5.) and no explicit indexing is allowed. Instead, parallel arrays can be manipulated as whole entities. All of the operators listed in Table 5 can be applied to parallel arrays; these are described in more detail in Section 5. Parallel arrays may be arbitrarily composed with indexed arrays and records, but they cannot be nested inside of other parallel arrays, i.e., the elements of a parallel array cannot be a parallel array.

The *rank* of a parallel array is its number of dimensions. There is no limit on the number of dimensions that a parallel array may have.[1]

[1]The current implementation of the compiler assumes no more than six dimensions.

3.3.3 Indexed Arrays

An indexed array is a *sequential* data type—its use can yield no parallelism. Indexed arrays are identical to arrays in languages such as Pascal and Modula-2. They *may* be indexed and their bounds must be known at compile time. In particular, their bounds may be expressions that are known at compile time, so they may involve constants but not config parameters (Section 4.1.1). The conversion of sequential variables to parallel variables is discussed in Section 6.2.

Note that there is a distinction between nested indexed arrays and multi-dimensional indexed arrays. They are declared differently and accessed differently. Below, `a` is a nested indexed array and `b` is a multidimensional indexed array.

```
var a : array [1..10] of array [1..5] of real;
    b : array [1..10, 1..5] of real;

a[1][5] := b[1,5];
```

3.3.4 Unions

Unions are similar to records except that memory is shared across the union's fields. Individual fields are accessed using the `.` operator. Unions may not have parallel variables as fields.[2]

3.4 Directions

Directions are vectors that have several uses in ZPL. They can be used to create new regions (see Section 5.4.7), and they can be used to shift a parallel array reference (see Section 5.1.1).

An example of a definition for a direction of rank two is shown below:

```
direction east      = [0, 1];
          northeast = [-1,1];
```

Directions may be defined in terms of constants or expressions involving constants. Like regions, directions are not first class objects. They cannot be modified and cannot be passed as parameters to procedures.

3.5 Regions

Regions are index sets that are used to define parallel arrays and define the domain for parallel array operations. Regions can be defined as follows:

```
region    R    = [1..N, 1..M];
```

[2]Unions are currently not supported by the compiler.

The bounds of a named region are specified in terms of integral expressions involving only constants and config variables.

Once defined, regions can be used to declare parallel arrays:

```
var       A, B: [R] real;
          I:    [R] integer;
```

Regions are applied to statements to define the domain over which the computation should occur. The following use of the region `[R]` indicates that each element of `B` in the index set `R` will be assigned to the corresponding element of `A`.

```
[R]          A := B;
```

All parallel arrays in a statement must have storage for the indices of the region that applies to it.

3.5.1 Of Regions

Regions can be defined relative to other regions using the `of` operator as shown below:

```
direction east = [0,1];
region    R    = [1..N, 1..M];
          E    = [east of R];
```

Note that `[E]` is also considered to be an "Of Region." Thus, the following two statements are equivalent.

```
[east of R] wrap A;
[E]         wrap A;
```

However, a subtle detail is that `[E]` and `[east of R]` differ in their effects on memory allocation. First, "raw" Of Regions, such as `[east of R]`, are not used when determining the data partitioning. This is consistent with the view that these regions serve primarily to define boundary conditions. By contrast, the indices in `[E]` *are* used to determine data partitioning. Second, implicit storage is only defined for "raw" Of Regions, not for named Of Regions.

```
[east of R] A := 0;        -- initialize east boundary of A
[E]         A := 0;        -- identical to above statement
```

3.5.2 Region Scope Rules

With one exception, a statement that refers to a parallel array must have a region of the same rank that applies to it. The exception is a procedure call that completely defines the size and rank of its parallel parameters and explicitly specifies regions for its statement body. For example, the following invocation of procedure `f` does not require a region.

```
procedure f (X : [R] integer);
[R]
begin
   . . .
end;

. . .

f(A);                          -- This statement does not require a region
```

Since a compound statement (see Section 7.2) is considered to be a statement, applying a region to a compound statement is equivalent to applying the region to each individual statement in the compound statement. This propagation of regions applies to all nested compound statements, as well. In the example below the region [R] applies to every statement.

```
[R] begin
      A := B;
      count := f(A);
      if count>0 then
          B := C;
      end;
    end;
```

The body of a procedure need not specify a region for every statement. In this case, the region scope that exists at the procedure's call site is propagated into the body of the procedure and applied to individual statements as needed. An example is shown in Section 4.2.3. It is an error if no region scope of the proper rank is applied to the call site.

Regions do not affect control flow (see Section 3.5.4). Regions can be viewed as providing index sets for expressions that require them. Thus, applying a region to a statement that has no parallel variables has no effect. Furthermore, a statement may have multiple regions of different rank applied to it; this is needed for operations that involve operands of different rank (e.g. partial reductions and scans; see Section 5.3) and is sometimes useful for compound statements.

3.5.3 Restrictions on Regions

Regions are not first class objects. They cannot be used with any operators other than `with`, `without`, and `of`. They cannot be assigned to, and they cannot be passed as parameters to procedures.

3.5.4 Regions Lists

Region lists provide a way to reduce code replication. The code below assigns the elements of `B` to corresponding elements of `A` inside of both `R` and `E`.

```
[R][E] A := B;
```

The above code is semantically equivalent to the following:

```
[R]    A := B;
[E]    A := B;
```

Region Lists are applied to compound statements one statement at a time. So the following two blocks of code are equivalent:

```
[R][E] begin
           A := B;
           A := 0;
       end;

       begin
[R]        A := B;
[E]        A := B;

[R]        A := 0;
[E]        A := 0;
       end;
```

Note that a Region List has no effect if applied to a statement with no parallel variables, and a Region List has no effect on a statement that already has a region of the proper rank specified. This last point is illustrated by the following two code fragments, which are equivalent:

```
[R][E] begin
           A := B;
   [T]     A := 0;
           B := 1;
       end;

begin
   [R][E]  A := B;
   [T]     A := 0;
   [R][E]  B := 1;
end;
```

Note the difference between a Region List and the use of multiple regions of different rank to a single statement:

```
    region    R      = [1..N, 1..M];
              V1, V2 = [1..N];

    [R][E]   A := B;             -- Region List: replicates code
    [R][V1]  A := B;             -- two regions of different rank: only
                                 --   the appropriate region is used
```

When multiple regions are applied to a statement *the regions must all have different rank.* When a Region List is applied to a statement *the regions must all have the same rank.* It is not legal to apply multiple Region Lists of different rank to a statement; if this is desired, multiple compound statements may be used:

```
    [R][E] begin
             [V1][V2] begin
                        -- statements...
                      end;
           end;
```

3.5.5 Dynamic Regions

Dynamic regions are anonymous regions whose value need not be known until runtime. In the example below, `i` and `M` are integers:

```
    procedure f(i: integer);
    begin
[i, 1..M]   A := B;              -- copy the i_th row of B to the i_th row of A
    end;
```

The bounds of a dynamic region are fixed before its corresponding body of code is executed. Thus, in the following procedure the dynamic region is set to the 3^{rd} row of the index space even though the variable `i` is modified in the body of the code.

```
    procedure g();
    var i: integer;
    begin
        i := 3;
[i, 1..M]
        begin
            i := i+1;            -- this does not affect the dynamic region
            A := B;              -- copy the 3_rd row of B to the 3_rd row of A
        end;
    end;
```

The bounds of a dynamic region can be arbitrary expressions that evaluate to integers. In particular, region bounds cannot be parallel arrays, so the following is illegal:

```
    var A : [R] integer;

    . . .
[i, 1..A]  B := C;             -- illegal because A is a parallel array.
```

The `of` operator may not be used in conjunction with dynamic regions.

Dynamic regions incur somewhat higher cost than statically defined regions, so static regions should used be whenever possible.

3.6 Type Conversion

Type equivalence in ZPL is by name equivalence, not structure equivalence.

ZPL follows the same type conversion rules as ANSI C. Generally speaking, smaller values may be converted to larger ones, but assigning larger values to smaller values may give unpredictable results. ZPL does not provide any facility for casting one data type to another.

3.7 Sequential vs. Parallel Variables

A *parallel* variable is any data structure that contains a parallel array, for example, a parallel array of records, or an indexed array of parallel arrays. A *sequential* variable is any data structure that does not contain a parallel array. Similarly, a *sequential statement* in any statement that makes no reference to parallel variables. As mentioned in Section 3.3.2, parallel variables are distributed across multiple processes, and thus are the basic source of parallelism. Sequential variables, on the other hand, are replicated across the processors, but ZPL semantics guarantee that the multiple replicas are kept coherent. By convention in this manual, all parallel variables have names that begin with an uppercase letter.

4 Variables

4.1 Special Variables

4.1.1 Config Variables

ZPL provides a special class of variables known as *config* variables, or config parameters. These are essentially write-once variables that can be specified at load time via command-line parameters.

There are three ways to initialize config variables. They may be specified at runtime by using the `-s` flag on the command line (see Section 9.2, they may be initialized in a file by using the `-f` flag on the command line, or they may be given default values when they're declared in the text of the program. The values specified at the command line override any other values, and those given in a file override any of the default values. Config parameters may be initialized in terms of expressions of constants or other config parameters.

4.1.2 "Index" Arrays

ZPL provides a set of special parallel arrays named `IndexRMO`, `Index1`, `Index2`, etc., These are constants that are defined by the compiler and conformable with any region whose rank is no greater than the index number (`IndexRMO` is conformable with any parallel variable). The elements of `IndexRMO` are guaranteed to be unique for each index regardless of the region that it is applied to. For the other "Index" Arrays, the value of each element of the i^{th} "Index" Array is the index of that element in the i^{th} dimension.

```
A := Index1;              -- assign row indices to each element of A
```

There are certain restrictions on the use of these parallel variables.

- "Index" Arrays cannot be modified.
- The rank of "Index" Arrays must be defined by their context. In particular, they cannot be used in I/O statements (see Section 7.4) unless additional context specifies their rank. For example, the following is legal because the array `A` defines the rank of `Index1`: `write(A+Index1);` The following is not legal because there is no context to indicate the rank of `Index1`: `write(A, Index1);`

4.2 Declarations

4.2.1 Scope Rules

Variables must be declared before they are used. Except for regions, which are dynamically scoped, ZPL uses two-level lexical scoping: Variables may be declared at the global scope or they may be declared within the scope of a procedure. Variables defined local to a procedure hide any global variables of the same name.

Variables are defined with the keyword `var`:

```
var a : real;           -- declaration of a sequential variable
    A : [R] real;       -- declaration of a parallel array
```

Multiple variables may be declared using the same type specifier:

```
var X, Y : [R] real;
```

Nested structures are declared from left to right. The following declares an indexed array of parallel arrays of real, that is, ten distinct parallel arrays of reals.

```
var B : array [1..10] of [R] real;
```

By contrast, the following declares a parallel variable where each element consists of ten reals.

```
var C : [R] array [1..10] of real;
```

Finally, the following example shows a parallel array where each element is a record:

```
type complex = record
       r : real;
       z : real;
     end;

var D : [R] complex;    -- parallel array of complex records
```

4.2.2 Declaring Degenerate Dimensions

The above example shows how indexed arrays and parallel arrays are declared by specifying upper and lower bounds for each dimension. A short-hand is provided for degenerate dimensions that contain only a single index. The following two declarations are equivalent.

```
region east = [1..N, N..N]
       East = [1..N, N];
```

This short-hand is legal for declaring indexed arrays, parallel arrays, or dynamic regions (see Section 3.5.5).

4.2.3 Rank Defined Arrays

At the global scope, parallel variables must be declared using regions, as shown above. However, parallel variables that are local to a procedure can be defined using only their rank. This gives the procedure the flexibility to accept parallel parameters of any region of the appropriate rank. Below, the procedures `Double` and `Swap` are defined for two-dimensional parallel arrays.

```
    procedure Double (X : [2] real) : [2] real;
    begin
        return X * 2;
    end;

    procedure Swap (var X,Y : [2] real);
    var Temp : [2] real;
    begin
        Temp := X;
        X    := Y;
        Y    := Temp;
    end;
    . . .

[east of R] Double(A);
```

```
[west of R] Double(A);
[R]         Swap (A, B);
```

Note that there are two concepts at work here. First, a parameter that is rank-defined can take any variable of the same rank as its actual parameters. Second, return values and local parallel arrays assume the size of the region scope that is specified at the call site.

4.2.4 Implicit Storage for Parallel Variables

Of Regions are viewed as boundary conditions, so storage is implicitly extended for parallel arrays that are modified in the context of an Of Regions. Implicit storage is only defined for the *base* region, i.e., the region with which the parallel variable was declared, and only for parallel variables that are modified in the Of Region. Examples are shown below.

```
    region  R = [1..N, 1..N];
            E = [east of R];

    var     A, B : [R] real;-- A and B declared to have the same base size

    . . .

[east of R] A := 1.0;       -- A's storage is implicitly extended to include
                            -- [east of R]
[E]         B := 1.0;       -- B's storage is implicitly extended.
```

Implicit storage is not used in determining the partitioning of a parallel variable.

Note that it is an error to reference a parallel variable outside of its defined storage. Thus, the following is illegal.

```
    var     A, B : [R] real;

    . . .

[east of R] A := 1.0;       -- implicit storage defined for A
    [R]     A := B@east;    -- error: no implicit storage defined for B
```

4.2.5 Procedure Prototypes

Before a procedure is invoked, it must either be defined or prototyped. A prototype is a header that defines a procedure's parameter and return types. Procedure prototypes are needed for defining mutually recursive procedures or for invoking external procedures written in C (see Section 8.2.

4.3 Initializing Variables

Config variables must be initialized when they are declared. Variables other than config variables must be initialized in the program text:

```
config var
    N : integer = 100;

var
    heat : real;        -- cannot initialize heat in its declaration
...

heat := 1.0;            -- initialize heat in the program text
```

5 Expressions and Operators

The set of ZPL operators is shown in Table 5 along with the associativity of each operator. Operators with highest precedence are listed at the top of the table. Those with lowest precedence are listed at the bottom. The `@` operator has highest precedence because it specifies an l-valued expression (see Section 5.6). Unary operators have higher precedence than the reduction and scan operators (which are can be either unary or binary operators), and these in turn have higher precedence than the remaining binary operators.

5.1 Logical Operators

ZPL's logical operators apply to *integral* operands (see Section 3.1). Zero represents logical false. Non-zero represents logical truth.

5.1.1 The At Operator

The At operator (`@`) takes two operands—a parallel variable and a direction—and results a parallel variable of the same size and shape, but displaced from the original by the direction vector. The parallel operand must have a legal l-value (see Section 5.6). The resulting expression itself has an l-value and can be used wherever a parallel variable is used. In particular, it can appear on either the lefthand side or righthand side of assignment statements, and it can be passed as a parameter to procedures.

5.2 Unary Operators

5.2.1 Unary Plus and Minus

Unary plus (`+`) and unary minus (`-`) are prefix operators that must be applied to arithmetic operands. The type of the resulting expression is the type of the operand.

Table 5: Operators in ZPL.

At Operator (left associative)

`@`

Unary Operators (right associative)

`+`	`-`	`!`	`~`

Reductions and Scans (right associative)

`+\`	`*\`	`max\`	`min\`	`and\`	`or\`	`&\`	`|\`	`^\`
`+\\`	`*\\`	`max\\`	`min\\`	`and\\`	`or\\`	`&\\`	`|\\`	`^\\`

Binary Operators (left associative)

		`*`	`/`	`%`	
		`+`	`-`		
		`>>`	`<<`		
`>`	`<`	`>=`	`<=`	`=`	`!=`
		`&`	`|`	`^`	
	`and`	`or`	with	without	`of`

5.2.2 Logical Negation

The logical negation operator (`!`) applies to integral operands and evaluates to an `integer` type.

5.2.3 Bitwise Complement

The bitwise complement operator (`~`) returns the one's complement of an integral operand.

5.3 Reductions and Scans

ZPL provides a set of reduction and scan operators that reduce a parallel variable to a lower-dimensional object (perhaps a sequential value). A reduction applies an operator to a parallel expression and produces a sequential expression. For example, the maximum reduction `max\` of a parallel expression evaluates to a sequential value that is the value of the largest element of the parallel expression.

```
[R] s := max\ A;                -- assign to s the largest value in A
```

These operators accept either one or two operands.[3] The reduce and scan operators can be applied to arbitrary parallel expressions. An optional parameter that immediately proceeds the operator is used to specify partial scans and reductions. For example, a reduction across the first dimension produces a parallel expression that is the same shape as the parallel operand but with its first dimension missing.

```
    V := +\ [1] A;              -- reduce the first dimension of A
```

A negative dimension indicates that the scan is to be performed in reverse order, i.e., from the largest index to the smallest for the specified dimension. (These dimension parameters may be specified for reduction operators, but the order of evaluation for reductions does not produce any semantic difference.)

The scan operators compute the parallel prefix of a parallel expression. Each element is the reduction of the values that precede it. The order of "precedes" is, by default, row major order. The order can be defined by the user by specifying the order of the dimensions as optional parameters. For example, the following performs a scan in the opposite order from the default.

```
    A := +\\ [-2][-1] A;        -- compute parallel prefix in reverse order
```

The `&&` and `||` symbols are synonyms for `and` and `or`, respectively. These synonyms can be used in reduce and scan operators, namely, `&&\`, `&&\\`, `||\`, and `||\\`.

[3]Future implementations will allow a third operand to specify segmented scans.

5.4 Binary Operators

ZPL's binary operators are all infix operators. When given parallel operands, binary operators are applied to corresponding elements of the two operands in an undefined order.

5.4.1 Arithmetic Operators

The arithmetic operators—addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`) and modulus (`%`)—accept arithmetic operands; the type of the resulting expression is the type of the larger of the two operands (See Section 3.6 for type conversion rules). Multiplication, division, and modulus have higher precedence than addition and subtraction.

The modulus operator must be applied to integral operands. It returns the remainder of the left operand divided by the right operand. When applied to integral operands, the division operator returns the quotient of the two operands.

5.4.2 Shift Operators

The shift operators—bitwise-right-shift (`>>`), and bitwise left shift (`<<`)—apply only to integral operands. The details of sign extension are machine-dependent and are defined by the target machine's C compiler. These operations are undefined if the second operand is negative.

5.4.3 Relational Operators

The relational operators—equality (`=`), inequality (`!=`), greater-than (`>`), less-than (`<`), greater-than-or-equal-to (`>=`), and less-than-or-equal-to (`<=`)—compare two arithmetic operands, returning an integral value.

5.4.4 Bitwise Operators

The bitwise operators—bitwise-exclusive-or (`^`), bitwise-and (`&`), and bitwise-or (`|`)—apply logical operators to corresponding bits of two integral operands.

5.4.5 Logical Operators

The logical operators—*and* (`and`) and *or* (`or`)—can be applied to arithmetic operands. *And* returns 1 if both operands are non-zero, and 0 if either operand is zero. *Or* returns 1 if either operand is non-zero, and 0 if both operands are zero. The precedence of *and* is higher than that of *or.* `&&` and `||` are synonyms for *and* and *or*, respectively.

5.4.6 Mask Operators

A mask is a region expression that specifies an index set based on the value of a parallel operand. The `with` operator produces an index set that is the intersection of the region and the true elements (non-zero) of the parallel operand. The `without` operator intersects with the false elements of the parallel operand. The following code fragment shows how a Red-Black SOR program could be written using masks.

```
    var  Mask : [R] integer;


    . . .
                                -- set mask to checkerboard pattern
    Mask := ((Index1 % 2)  and  (Index2 % 2) or
             !(Index1 % 2) and !(Index2 % 2));
[R with Mask]    Sor(A);
[R without Mask] Sor(A);
```

The `with` and `without` operators can take one or two operands. The mask operand must always be specified, but the region may be omitted, in which case the existing region scope is used to compute the intersection. Thus, the following two statements are equivalent.

```
[R]               begin
[with Mask]           Sor(A);
[without Mask]        Sor(A);
                  end;

                  begin
[R with Mask]         Sor(A);
[R without Mask]      Sor(A);
                  end;
```

The mask operand must be a parallel variable with an l-value (see Section 5.6). This restriction simply prevents the use of complicated expressions as the mask operand.

5.4.7 The Of Operator

The Of operator takes two operands—a region and a direction—and evaluates to a new region that is adjacent to the original region. The relative position of the new region is determined by the signs of the components of the direction vector, and the size of the new region is determined by circumscribing a box around the direction as shown in Figure 1.

Mathematically, if vector v = $(v_1, v_2, ...v_n)$ and
region R = $(l_1, u_1) \times (l_2, u_2) \times ... \times (l_n, u_n)$,

[v of R] defines an array of size $v'_1 \times v'_2 \times ... \times v'_n$,
where $v'_i = \mid v_i \mid \quad$ if $v_i \neq 0$,
or
$v'_i = u_i - l_i + 1 \quad$ if $v_i = 0$.

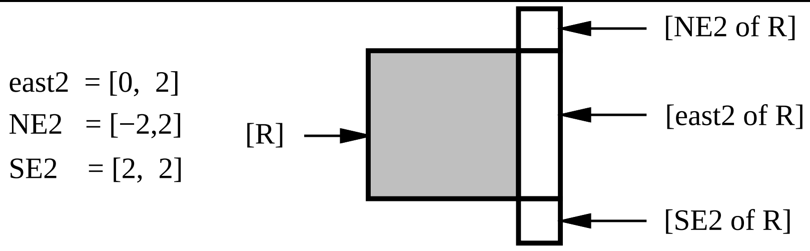

Figure 1: Illustration of "Of" Regions.

This array is located such that it is adjacent to R at $(r_1, r_2, ...r_n)$,

$$\begin{array}{ll} \text{where } r_i = l_i & \text{if } v_i < 0 \\ \text{or} & \\ r_i = u_i & \text{if } v_i > 0. \end{array}$$

In other words, if $v_i < 0$, the upper bound of the Of array is the lower bound of the i^{th} dimension. If $v_i > 0$, the lower bound of the Of array is the upper bound of the i^{th} dimension.

5.5 Procedure Calls

By default, parameters in ZPL are passed by value, which means that any modifications to the formal parameter do not affect the actual parameter. Note that this may be expensive because parameters may have to be copied to preserve "call-by-value" semantics. Parameters can be passed by reference by prefixing the `var` keyword to the formal parameter declaration, in which case any modifications to a formal parameter will also affect its corresponding actual parameter. Below, the first parameter is passed by reference and the second is passed by value.

```
    procedure p (var A: [R] real; B: [R] real);
```

When nested data structures are passed by value, only the top-level is copied. For example, if a record is passed by value, the record itself is copied but its subfields are not.

Recursion is allowed.

5.6 L-Values

Certain expressions refer to a user-defined data object and are said to have *l-values*. Such expressions can appear anywhere that a simple variable name can be used. In particular, such expressions can appear on the lefthand side of assignment statements. In addition, if a formal parameter is passed by reference, its corresponding actual parameter must have an l-value. Both of these cases require l-values because they specify the modification of a memory location.

An expression has an l-value if it is the name of a variable or if it is uses only the following constructs: indices ([ ]), record fields (.), and Ats (@).

6 Sequential and Parallel Variables

This section describes additional features of indexed arrays that are particularly useful when indexed arrays are components of parallel variables. This section also explains how sequential variables can interact with parallel variables through *promotion*.

6.1 Operations on Whole Indexed Arrays.

A limited set of aggregate operations are provided for indexed arrays that allows them to be manipulated as whole entities. These are the assignment statements (see Table 6) and the binary operators (see Section 5.4). For example, the following is legal:

```
var a, b : array [1..10, 1..5] of real;
    c    : real;

a := b;             -- aggregate assignment allowed for indexed arrays
```

Unlike parallel arrays, whole indexed arrays cannot appear as operands to reduction or scan operators, cannot be promoted (see Section 6.2) and do not result in shattered control flow (see Section 7.2.4). However, whole indexed arrays *can* be used in the `wrap` and `reflect` statements (Section 7.3 describes `wrap` and `reflect`) if these indexed arrays appear in parallel variables. This rule may seem strange, but it makes sense because `wrap` and `reflect` are special forms of the assignment statement. The restriction that the indexed arrays appear inside a parallel variable is needed because `wrap` and `reflect` specify operations on "Of Regions," and regions have no effect on purely sequential statements (see Section 3.7).

Note that a single element of an indexed array can be specified or an entire indexed array can be specified, but there is no way to specify a sub-portion of an indexed array. This "all or nothing" rule applies to each individual indexed array, and is illustrated below using the following declarations:

```
type array1 = array[1..10] of integer;
     array2 = array[1..5]  of array1;
var  a, b : array1;
     c    : array2;

a := b;             -- all elements of 'a' and 'b' specified
c[1] := a;          -- one element of 'c' and all of 'a' specified
```

A more complicated example is shown below using the following declarations:

```
direction east = [0,1];
region    R    = [1..N, 1..N];
var a, b : array [6..10] of [R] array [1..5] of real;
```

The following two statements manipulate an indexed array of parallel arrays of indexed arrays as a single entity.

```
a := b;
a@east := b@east;
```

The statement below specifies the entire outermost indexed array but only a single element of the inner indexed array; this is possible because the use of the `@` operator explicitly differentiates the indices of the outer indexed array from those of the inner nested array.

```
a@east[1] := b@east[1];
```

In the absence of such a disambiguating symbol, the indices are bound beginning from the outermost indexed array. Thus, the following two statements specify a single element of the outer array and all of the inner array.

```
a[10] := b[10];
a[10]@east := b[10]@east;
```

6.2 Promotion

Sequential and parallel expressions interact through promotion. In the following example the sequential variable `1.0` is logically promoted to be a parallel array of the same size as `A`.

```
A := 1.0;                    -- promote 1.0 to a parallel array
```

Similarly, the return value of a procedure can be promoted by assigning it to a parallel variable:

```
var A : [R] real;
    s : real;

procedure abs(x: real) : real;
begin
    if x<0 then
        return -x;
    else
        return x;
end;
```

```
    . . .

    A := abs(s);               -- promote the value returned by abs()
```

It is illegal to modify a promoted sequential variable.

```
    s := A;                    -- illegal promotion of s
```

6.2.1 Parameter Promotion

Parameters may be promoted by passing in a sequential actual where a parallel formal is expected. This is analogous to the above case where a sequential was promoted in an assignment statement. It is an error to promote a sequential parameter that is modified in the body of the procedure and is passed by reference.

6.2.2 Procedure Promotion

Procedures can be promoted by passing in parallel parameters where sequential parameters are expected. The procedure is logically applied to each element of its parallel parameters.

```
    procedure abs (x : real) : real;
    begin
        if x < 0 then
            return -x;
        else
            return x;
    end;

    . . .
[R] A := abs(A);               -- promote the abs() procedure
```

The remainder of this section explains two restrictions on the promotion of procedures: *Procedures that refer (directly or indirectly) to parallel procedures cannot be promoted; and procedures with side effects cannot be promoted.*

Only *sequential procedures*—those procedures whose bodies do not refer to any parallel variables—can be promoted. Procedures that refer to a parallel variable are known as *parallel procedures*. Note that a procedure that invokes a parallel procedure is itself a parallel procedure, and note that all I/O procedures are considered to be parallel procedures. The following code fragment shows an illegal attempt to promote a parallel procedure.

```
    procedure Scale (X : [R] real, factor: integer) : [R] real;
    begin
        return X * factor;
    end;

    . . .
[R] A := Scale (A, B);        -- illegal promotion of a parallel procedure
```

Procedures with side effects should not be promoted because the order in which the procedure is applied to the elements of the parallel parameters is not well-defined. To understand this, consider the following promoted procedure.

```
    procedure copy (var a, b: real);
    begin
        a := b;
    end;

    . . .
[R] copy (A, A@east);         -- dangerous use of promotion
```

The above procedure is applied to corresponding elements of `A` and `A@east` in the region `[R]`. However, because the procedure is applied to its parameters in an unspecified order, the results are unpredictable. By contrast, the following parallel procedure has well defined semantics because the right hand side of the statement is guaranteed to be evaluated before it is assigned to the left hand side (see Section 7.1).

```
    procedure Copy (var A, B: [2] real);
    begin
        A := B;
    end;

    . . .
[R] Copy (A, A@east);
```

7 Statements

ZPL statements are executed sequentially. Logically, a statement does not begin execution until the previous one completes.

7.1 Assignment Statements

Table 6 shows the ZPL assignment statements. The right hand side of an assignment statement is evaluated before it is assigned to the left hand side. The operator-assignment statements, such as `A += B`, are shorthand for `A := A + B;`

Table 6: Assignment Statements in ZPL.

`:=`
`+=`
`*=`
`-=`
`/=`
`%=`
`<<=`
`>>=`
`&=`
`|=`
`^=`

7.2 Compound Statements

A ZPL statement may consist of a compound statement. The keywords `begin` and `end` are used to form a compound statement from a group of statements; and any regions that apply to this compound statement apply to each of the individual statements. The other types of compound statements are shown in Table 7, where the square brackets represent an optional item, and the square brackets followed by an asterisk represent zero or more instances of an item.

7.2.1 Unconditional Control Flow

There are four statements that specify unconditional control flow: `halt`, `continue`, `exit`, `return`. The `halt` statement stops program execution. The `continue` and `exit` statements are used in iterative control constructs. The former skips to the next iteration of the nearest enclosing loop and the latter exits the nearest enclosing loop. Finally, the `return` statement terminates a procedure call and returns control to the calling procedure. For procedures with return values, `return` is followed by an expression that is the return value.

7.2.2 Conditional Control Flow

The `if` statement evaluates its conditional *expression* and executes the `then` *statements* if the conditional is true. If the conditional evaluates to false, the optional `elsif` clause (of which there may be any number) is tested and executed. If none of these clauses evaluates to true, the optional `else` *statements* are executed.

Table 7: Compound Statements in ZPL.

```
begin
     statements;
end;

if   expression then
     statements;
[elsif expression then
     statements;]*
[else
     statements;]
end;

for  index := initial to final [by step] do
     statements;
end;

for  index := initial downto final [by step] do
     statements;
end;

while expression do
     statements;
end;

repeat
     statements;
until expression;
```

7.2.3 Iterative Control Flow

The `for`, `while` and `repeat` statements specify iteration. The `for` statement's *index* variable is initialized to *initial* and is compared against the *final* value before each iteration. The `for`...`to` statement increments the value of the *index* variable by one after each iteration and terminates when the value of *index* is greater than *final*. The `for`...`downto` statement decrements the value of the *index* variable by one after each iteration and terminates when the value of *index* is less than *final*. The optional *step* expression specifies the amount of the increment or decrement; *step* must be an integral expression that is greater than zero (negative or zero values of *step* can lead to infinite loops). It is legal to modify the *index* value inside the body of the statement.

The `while` statement evaluates its conditional *expression* before each iteration of its loop body. The loop terminates when the *expression* evaluates to false. The `repeat` statement is similiar to the `while` except it evaluates its conditional *expression* at the end of each iteration.

7.2.4 Shattered Control Flow

The statements described earlier in this section have all provided a single thread of control in that a single statement executes at a time. ZPL also has the notion of *shattered control flow*: When a parallel expression is used in the conditional of a compound statement, the control flow is said to *shatter* and each index value has its own thread of control. Shattered control flow is analogous to an anonymous sequential procedure that is promoted by passing parallel variables as parameters.

Certain restrictions are placed on shattered control flow to guarantee that when control flow shatters, the results are predictable and meaningful.

- Sequential variables cannot be modified inside shattered control flow. This would be analogous to promoting a sequential procedure that has side effects.
- Each thread may only refer to parallel elements with the same index. Thus, the use of `@`, `wrap`, `reflect`, reductions and scans is illegal.
- Parallel variables of only a single rank can be used inside shattered control flow.

Note that shattered control flow still provides sequential semantics. Nested shattered control flow is allowed, but they must all have the same rank. The "Index" Arrays, such as `Index1` and `IndexRM0`, are allowed in shattered control flow.

7.3 Wrap and Reflect

ZPL's `wrap` and `reflect` statements are used to initialize periodic and mirrored boundary conditions, respectively. The example below initializes the parallel array `A` in the region `[east of R]` by reflecting the values in `[R]` across the boundary between `[east of R]` and `[R]`.

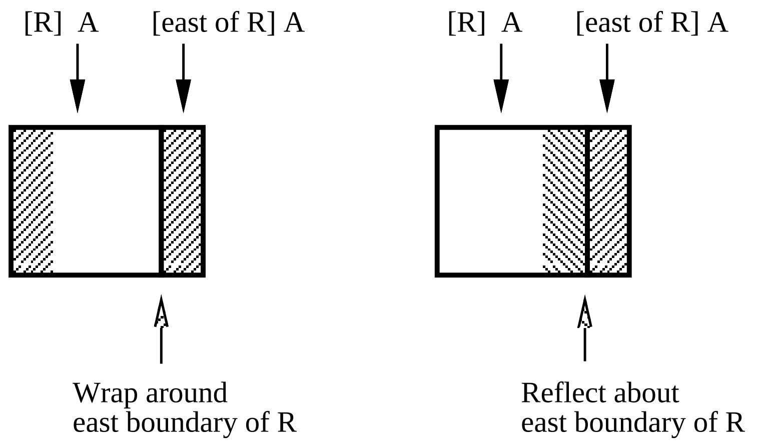

Figure 2: The Reflect and Wrap Operators.

```
direction  east = [0, 1];
region     R    = [1..N, 1..N];
var        A    : [R] real;

[east of R] reflect A;
[east of R] wrap A, B;
```

The `wrap` statement is similar to `reflect` except it copies data from the western columns of `R` as if the region were connected as a torus. (See Figure 2.)

The operands to `wrap` and `reflect` must be parallel variables. The boundary between the source and destination regions must be well defined, so these statements must have an Of Region applied to them. Note that a named region that is defined using Of is itself considered an Of Region. If an Of Region is a defined in terms of some other Of Region, as shown below, the boundary used by `wrap` and `reflect` is defined by the outermost Of, in this case the boundary between `[E]` and `[EE]`. (See Figure 3.)

```
region     E  = [east of R];
           EE = [east of E];

[EE]       reflect A;
```

Since `wrap` and `reflect` are forms of assignment statements, they implicitly define storage for their operands when used in Of Regions, as explained in Section 4.2.4.

Note that `wrap` and `reflect` may be applied to operands involving `@`'s, so the following is legal:

```
[east of R]  reflect X@west;
```

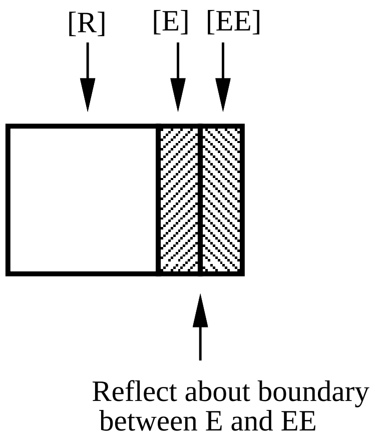

Figure 3: Reflect for Nested Of Regions.

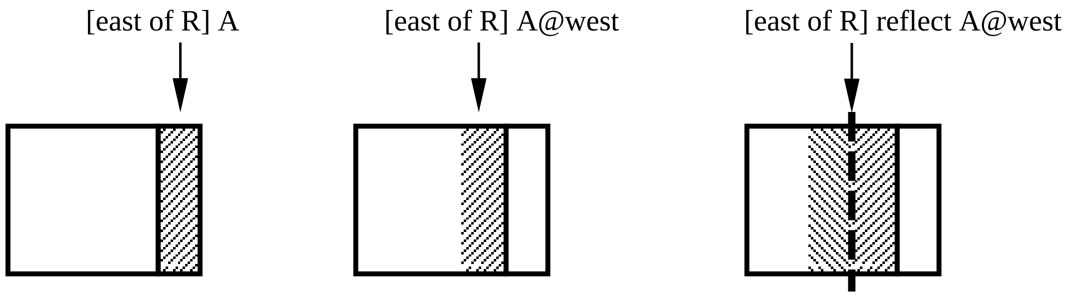

Figure 4: Illustration of Reflect Applied to Operands with @'s.

In this case `[east of R] X@west` is the target of the reflect, and the source is the data reflect across the boundary between `[east of R]` and `[R]` shifted `west`. This is illustrated in Figure 4.

7.4 Input and Output

ZPL provides basic facilities to open and close files, to read from and write to files, and to customize I/O for parallel data structures.

7.4.1 Open and Close

Before a file is accessed it must be opened using the `open()` statement, and after the last access the file is closed using `close()`. The `open()` routine takes two parameters. The first is a string that gives the name of the file to open. The second is a string that specifies the mode of access: `"r"` for read, `"w"` for write, and `"a"` for append. If successful, `open()` returns a `file` variable that can be used to access the file as described in the next subsection. If unsuccessful, `open()` returns `0`.

The following example shows how a file is opened for reading.

```
var f : file;
. . .

f := open ("input", "r"); -- open file for reading
```

The `close()` procedure takes a `file` parameter that was obtained from `open()`.

7.4.2 Read and Write

ZPL's `read` and `write` statements are used to transfer typed variables to and from files. The `read` and `write` statements have the following syntax, where *file* is an optional `file` parameter and *exprls* is an arbitrary list of expressions.

```
read    ([file,] exprls);
write   ([file,] exprls);
writeln([file,] exprls);
```

The `file` parameter specifies the file to access. If this parameter is not specified, input comes from `stdin` and output goes to `stdout` (`stdin` and `stdout` are always open and never need to be closed). For `read()`, the list of expressions is a list of variables to read into. For `write()`, the list of expressions is a list of variables to write. These expressions may be either sequential or parallel variables. If any are parallel variables a region must apply to the statement to specify the portions of the parallel variable to read or write. The `writeln()` procedure is identical to `write()` except it writes a newline character after writing its expression list.

```
    var f : file;
        i : integer;
        A : [R] real;
    . . .
    read(f, i);                 -- read an integer into variable i
[R] read(f, A);                 -- read a parallel of reals into variable A
    close (f);                  -- close the file
```

The `read();` and `write();` statements take a variable number of expressions, so the above two `read();` statements could have been written as one:

```
[R] read(f, i, A);              -- equivalent to the two reads() above
```

Parallel variables are serialized when they are written to a file. Thus, the file contains a sequence of elements and *all region structure is lost.* This means that if a parallel is written to file using one region and is then read from the file using a subset of that region, the values that are read will not correspond to those in the subregion of the original parallel variable. Similarly, care must be taken when applying masked regions to I/O statements (Section 5.4.6 describes masks).

7.4.3 Control Strings

The format of each expression may be controlled using a control string that precedes the expression and is separated by a colon (`:`). These control strings use the same structure as the control strings in C `printf()` routines. For example, the following specifies that each element of the parallel array `A` will be printed to 5 decimal places.

```
    [north of R]  write("The north boundary contains ", "%.5f": A);
```

Note that the control string applies to each element of the parallel array. Thus, the following statement differs from the above in that the following will print `"The north boundary contains"` once for each element in `[north of R]`, whereas the above statement will print the string just once.

```
    [north of R]  write("The north boundary contains %.5f": A);
```

Control strings also may be applied to sequential expressions. The colons are used strictly to separate control strings from expressions, so each colon must follow a string and precede an expression.

7.4.4 Customized I/O Routines

There are two restrictions on reading and writing parallel variables that can be corrected by using customized I/O routines. First, parallel variables are by default read or written one element at a time; customized I/O routines provide a way to print these two fields side by side. Second, these routines provide a way to print derived data types as a single expression.

For example, if `People` is a parallel array of records, the following statement will first write all of the `name` fields followed by all of the `age` fields.

```
[R] write(People.name, People.age);
```

A customized I/O routine is a procedure that specifies how to print a single element of a parallel variable. Once bound to a parallel variable using `bind_write_func()`, subsequent I/O calls will use the user-supplied procedure to print each element of the parallel variable . Thus, the following will cause the `name` and `age` fields to be printed side by side:

```
procedure Write_Person(outfile: file; var p: person);
begin
    write(outfile, p.name, p.age);
end;

. . .
bind_write_func(People, Write_Person);
[R] writeln(People);
```

The first parameter to `bind_write_func()` is a `file` parameter. The second parameter is one element of the parallel variable. *This must be a* `var` *parameter but must not be modified inside the procedure body.*

The customized print routine can be unbound using `unbind_write_func()` with the parallel variable's name as a parameter:

```
unbind_write_func(People);
```

A pair of analogous routines, `bind_read_func()` and `unbind_read_func()`, are supplied for reading parallel variables.

7.4.5 Caveats

Customized I/O routines can only be bound to parallel variables, that is, a single identifier that contains no array indexing, no record field names, and no `@`'s. For example, `A` is a legal variable name but `A.x` is not.

I/O statements are considered to be parallel procedures because the multiple processes must coordinate to print expressions—both sequential and parallel—in a coherent manner. Thus, it is illegal to promote `read()`, `write()`, or `writeln()`. This is illustrated below:

```
var  A : [R] real;
procedure print_real (var a: real);
```

```
begin
    write(a);
end;

. . .
print_real(A);            -- illegal promotion of a parallel procedure
```

8 Program Structure

A ZPL program consists of the following items, which will be discussed in turn.

- a single ZPL module containing all ZPL source code
- C files that will be linked in
- object files that will be linked in
- external libraries that will be linked in

8.1 The ZPL Module

The ZPL compiler does not currently support separate compilation, so all ZPL code must reside in a single file (files may be included using the C pre-processor—see Section 9.1), but the compiler sees a single source file). The ZPL source file has the following syntax.

`program` *Identifier*;
Declarations
Procedures

The program's name, *Identifier*, must be a legal ZPL identifier (see Section 2.4.1). *Declarations* represents the set of globally defined config variables, constants, types and variables, as well as the program's directions and regions, which can only be defined at the global level. *Declarations* consist of any number of the following items in any order, where curly braces indicate an item that may be repeated zero or more times.

`config var` *Identifier*: *Type* = *Expression*;
{*Identifier*: *Type* = *Expression*;}

`constant` *Identifier*: *Type* = *Expression*;
{*Identifier*: *Type* = *Expression*;}

`type` *Identifier* = *Type*;
{*Identifier* = *Type*;}

```
direction    Identifier = [Vector];
            {Identifier = [Vector];}

region       Identifier = [DimensionList];
            {Identifier = [DimensionList];}

var          IdentifierList: Type;
            {IdentifierList: Type;}
```

Procedures represent procedure declarations and procedure prototypes. The general form of a procedure body is shown below. A procedure prototype consists of just the *Procedure-Header*.

```
procedure   Identifier ( { {var} IdentifierList: Type}) {: Type };
LocalVars
Statement
```

The local variables (*LocalVars*) of a procedure have the same form as the global variable declarations shown above, and the body is simply a ZPL statement (typically a compound statement using the `begin` and `end` construct).

One procedure must have the same name as the program name. This procedure is the program's entry point.

8.2 External Procedures

ZPL programs can invoke procedures written in C and other languages that use the same procedure call interface and object file format.

Because C allows data types that are not found in ZPL, two special data types are provided exclusively for declaring prototypes of external C procedures: `procedure` and `region`. These data types allow procedures and regions to be passed as parameters to C procedures.

ZPL programs treats sequential and parallel C procedures differently. Sequential procedures are treated just as sequential ZPL procedures. Thus, they can be promoted and any regions or masked regions will correctly apply to them. Any external procedure that is declared to accept parallel variables is considered to be a parallel procedure. *Regions and masked regions have no effect on external parallel procedures.* The assumption being that if an external procedure expects a parallel parameter, that procedure will itself be managing parallelism.

9 The Compiler and Runtime System

9.1 Compiler Flags

The ZPL compiler produces object C files along with a Makefile for compiling the resulting C code and linking them in with the machine-specific libraries. The machine-specific executable can be created by simply typing `make` in the proper directory. The current ZPL compiler, `zpl`, supports the following flags.

`-a`	Enables efficient access to parallel arrays. Program execution time will decrease while code size will increase.
`-c`	Enables optimized communication insertion algorithm. Both execution time and code size will decrease.
`-f`<file>	Specifies the ZPL source file.
`-h`	Displays this message.
`-i`	Reduces code size by not inlining communication code.
`-n`	Disables the C pre-processor. By default, the ZPL input file is passed through cpp before it is passed to the ZPL compiler.
`-p`<pass-file>	Specifies the pass file that determines the order in which the compiler's passes are invoked. Defaults to `pass.default`.
`-v`	Displays version information and quits.
`-I`<path>	Adds the specified include path during object code compilation, where object code refers to the C code produced by the ZPL compiler.
`-L`<path>	Adds the specified library during object code compilation.
`-l`<library>	Links in the specified library during object code compilation.

User-written C files and object files (files using the Unix `.o` format) can be also be specified at the command line when compiling a ZPL program. These C files an object files are included in the automatically generated Makefile. See the on-line documentation for further details on using the compiler and compiling the object C files.

9.2 Runtime Flags

The following flags can be supplied at runtime to the executable ZPL program. The `-p`, `r` and `c` flags specify the processor mesh configuration. If only the `-p` flag is specified, a default mesh configuration is used that is made as square as possible. If any two of these three flags are specified, the mesh will be configured as defined by the user.

The `-s` flag is used to set the value of config parameters.

`-c`<number>	Specifies the number of columns of processors to use.
`-f`<file>	Specifies a file from which to initialize config variables.
`-p`<number>	Specifies the number of processors to use.
`-r`<number>	Specifies the number of rows of processors to use.
`-s`<var>`=`<value>	Initializes the value of a config variable.

A ZPL Syntax

```
Key:    {  foo  }   foo may be repeated one (1) or more times
        {{ foo }}   foo may be omitted
        foo-LIST    stands for a comma-separated list of one or more foo's.
        foo-ID      is an identifier (ID) that is semantically a foo.
        Note that "[" and "]" are literals.


PROGRAM:        program ID; { DEFINITION }

DEFINITION:     direction      { ID = [ INTEGER-LIST ]; }
        |       type           { ID = TYPE; }
        |       region         { ID = REGIONDEF; }
        |       var            { ID-LIST : TYPE; }
        |       config var     { ID-LIST : TYPE = INIT; }
        |       constant       { ID-LIST : TYPE = INIT; }
        |       prototype ID ( {{ FORMALS }} ) {{ : TYPE }} ;   -- procedure prototype
        |       procedure ID ( {{ FORMALS }} ) {{ : TYPE }} ;
                        {{ var { ID-LIST : TYPE; }  }}          -- local vars
                        STMT

TYPE:           integer        |       unsigned integer
        |       char           |       unsigned char
        |       shortint       |       unsigned shortint
        |       longint        |       unsigned longint
        |       real
        |       double
        |       TYPE-ID
        |       ( ID-LIST )                                     -- enumerated type
        |       array [ DIMENSION-LIST ] of TYPE                -- indexed array
        |       record { ID-LIST : TYPE; } end                  -- record
        |       union  { ID-LIST : TYPE; } end                  -- union (variant record)
        |       REGION  TYPE                                    -- array
        |       [ integer ] TYPE                                -- rank-defined array


DIMENSION:      EXPR .. EXPR
        |       EXPR                                            -- allows [1, 1..M]

INIT:           EXPR
        |       [ INIT-LIST ]

REGION:         [ REGION-ID ]
        |       [ DIRECTION-ID  of  REGION-ID ]

REGIONDEF:      [ DIMENSION-LIST ]
        |       { REGION }

MASK:           [ {{REGION-ID}} with    VARIABLE-ID ]
        |       [ {{REGION-ID}} without VARIABLE-ID ]
```

```
FORMALS:        -- a semicolon-separated list of one (1) or more FORMAL's
FORMAL:         {{ var }}  ID-LIST : TYPE

UN_OP:          one of: + - ~ !
BIN_OP:         one of: + - * / % << >> < <= > >= = != & | ^ and && or ||
ASSIGN_OP:      one of: := += -= *= /= %= <<= >>= &= ^= |=
SCAN_OP:        one of: +\\ *\\ min\\ max\\  and\\ &&\\  or\\ ||\\ &\\ |\\ ^\\
REDUCE_OP:      one of: +\  *\  min\  max\   and\  &&\   or\  ||\  &\  |\  ^\

PRIMARY:        VARIABLE-ID
        |       INTEGER-CONSTANT
        |       REAL-CONSTANT
        |       CHARACTER-CONSTANT                              -- e.g. 'c'
        |       { STRING-CONSTANT }                             -- strings concatenated
        |       ( EXPR )
        |       PROCEDURE-ID ( {{ EXPR-LIST }} )                -- procedure call
        |       PRIMARY [ EXPR-LIST ]                           -- indexed array
        |       PRIMARY @ DIRECTION-ID                          -- At operator

EXPR:           PRIMARY
        |       UN_OP     EXPR
        |       EXPR      BIN_OP    EXPR
        |       SCAN_OP   {{ [ INTEGER-LIST ] }} EXPR           -- []'s are for partial scans

        |       REDUCE_OP {{ [ INTEGER-LIST ] }} EXPR           -- scans and reduces

STMT:           REGION-LIST  STMT                               -- region specification
        |       MASK         STMT                               -- region with mask
        |       begin {STMT} end;
        |       PRIMARY ASSIGN_OP EXPR;
        |       PROCEDURE-ID ( {{ EXPR-LIST }} );               -- procedure call
        |       if    EXPR then {STMT} {{ { elsif EXPR then {STMT} } }}
                                       {{ else              {STMT} }}  end;
        |       while EXPR do {STMT} end;
        |       repeat {STMT} until EXPR;
        |       for ID := EXPR to     EXPR {{ by EXPR }} do {STMT} end;
        |       for ID := EXPR downto EXPR {{ by EXPR }} do {STMT} end;
        |       exit;                                           -- exit block
        |       continue;                                       -- go to next loop iteration
        |       halt;                                           -- stop program
        |       return {{ EXPR }} ;
        |       ;                                               -- null statement
        |       wrap    ID-LIST;
        |       reflect ID-LIST;
        |       read    (   EXPR-LIST   );
        |       write   (   EXPR-LIST   );
        |       writeln ( {{EXPR-LIST}} );
```

There are a few white lies to make this more readable: an extraneous comma is ignored at the end of INIT lists and enumerated type lists; enumerated type ID's may be given a value, e.g. "(R=2,G=4,B=8)"; and "+4" is a legal integer when defining directions or specifying which dimensions to scan/reduce.

Index