PADL — A Packet Architecture Description Language

A Preliminary Reference Manual

by

Clement K.C. Leung

William Y-P. Lim

October, 1983

The language design reported herein was supported by the National Science Foundation under research grant 7915255-MCS.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Cambridge Massachusetts 02139

PADL – A Packet Architecture Description Language

A Preliminary Reference Manual

by

Clement K.C. Leung

William Y-P. Lim

Abstract

PADL is a hardware description language for specifying the behavior and structure of packet communication systems. In such systems, hardware units called modules communicate by sending and receiving packets. The behavior of such a system can be specified by providing the algorithm it executes and the data structures it manipulates. On the other hand, the structure of a system is specified by giving the components or of the system and their interconnection. These components can be further specified structurally or behaviorally. The language constructs of PADL fall into two categories – those for behavior specification and those for structure specification. All these constructs which include the usual control constructs like conditionals and iterations, constructs for the packet oriented inter-module communication operations including a non-deterministic input operation, and facilities for data structuring, defining and invoking procedures, as well as for specifying, using and connecting modules, are described in this preliminary reference language manual.

Keywords: Hardware description language, packet communication systems.

CONTENTS

PADL – A Packet Architecture Description Language

1. INTRODUCTION

PADL (Packet Architecture Description Language) is a hardware description language for specifying the behavior and structure of packet communication systems. In a packet communication system, hardware is organized into modules which operate concurrently and communicate only by sending information packets to each other. This organization principle is particularly appropriate for modular, highly concurrent, hardware systems. The application of this principle has been illustrated in the design of a memory system [2] and in the design of highly parallel data-driven processor architectures [3]. PADL is intended for use as a hardware specification tool in the construction of practical hardware systems based on these designs.

In PADL a module can be specified from either a behavioral or a structural point of view. Behaviorally, a module receives input packets, processes them and generates output packets. The behavior of a module is specified by giving the algorithm it executes, and the data structures it manipulates. Structurally, a module is constructed as an interconnection of components. The structure of a module is specified by listing its components and the data paths between them. By providing for structural description, PADL explicitly recognizes structural composition and decomposition as important techniques in hardware design, and allows its user to develop hierarchical descriptions based on these techniques directly.

The language constructs in PADL fall naturally into two categories, for describing module structure and module behavior. The interconnection of discrete components at many different levels of detail in hardware design can be conveniently described using the structure description facilities provided in PADL. For behavior description, PADL contains the usual repertoire of control constructs, such as conditionals and iterations, data structuring facilities, procedure definition facilities, and intermodule packet communication facilities. In addition PADL also contains a simple construct for specifying nondeterministic operation and a rich set of primitives for bit strings. Noticeably absent from PADL are constructs for specifying concurrent operations within a module. Activities in different modules, however, may proceed concurrently and are synchronized by exchanging information packets. Thus the only mechanism for introducing concurrency in a PADL description is to decompose a module into an interconnection of concurrently operating submodules, while preserving its input/output behavior.

A hardware designer uses PADL to define procedures, complex data types, and new module types. He/she develops a library of definitions which he/she may then extend by either refining or building upon these definitions. This library is organized into a number of textual units each of which may contain one or more of these definitions. A definition may also refer to definitions in other textual units. Each such reference must be accompanied by enough typing information to allow complete type checking within each individual textual unit. This reference manual explains the language constructs provided in PADL for defining procedures, data types and modules. The organization of a designer's library of definitions, and the mechanisms for associating external references in a textual unit with other textual units in such a library are left unspecified.

1.1 Acknowledgements

PADL builds upon the work of several language development projects at the M.I.T. Laboratory for Computer Science and at the M.I.T. Lincoln Laboratory. Its structure description facilities are closely related to those provided in HISDL [6] and its behavior description facilities are based on those provided in ADL [5], HEX [4], and VAL [1]. Its data type facilities are based on those provided in CLU [7, 8].

We acknowledge the contribution of Professor Jack Dennis whose support, comments and criticisms have been useful during the design of the language and the preparation of this manual. The mechanism for handling values in PADL is borrowed from VAL, developed by him and Bill Ackerman. The latter has been particularly helpful in the preparation of this manual and his help is deeply appreciated. Some sections of this manual are adaptations of his work on VAL. The preparation of this manual has also been influenced by the feedback from the first users of PADL: they are Tam-Anh Chu, Guang-Rong Gao and Cindy Gilbert. Jim Holderle through his work on the PADL parser, has been very helpful in preparation of this manual by pointing out the bugs in the drafts of this manual. Others who have provided helpful comments and criticisms are Ken Todd and Tom Wanuga.

1.2 References

[1] W. B. Ackerman, and J. B. Dennis, *VAL – A Value-oriented Algorithmic Language preliminary reference manual*, Technical Report MIT/LCS/TR-218, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts, June 1979.

[2] J. B. Dennis, "Packet communication architecture," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, IEEE (August 1975), 224-229.

[3] J. B. Dennis, and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *Proceedings of the Second Annual Symposium on Computer Architecture*, IEEE (January 1975), 126-132.

[4] J. B. Dennis, *HEX – Hardware EXperimental description notation*, Course Notes for 6.823, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Massachusetts, Spring 1981.

[5] C. K. C. Leung, "ADL – An Architecture Description Language for packet communication systems," Computation Structures Group Memo 185, Lab. for Computer Science, M.I.T. October, 1979. An abbreviated version of this paper is published in *Proceedings of the 1979 International Symposium on CHDL and their Applications*, IEEE (October 1979), 6-13.

[6] W. Y-P. Lim, "HISDL – A structure description language," *Communications of the ACM 25*, 11 (November 1982), 823-830.

[7] Liskov, B.H., et. al., "Abstraction mechanisms in CLU," *Communications of the ACM 20*, 8 (August 1977), 564-576.

[8] Liskov, B.H., et. al., *CLU reference manual*, Technical Report MIT/LCS/TR-225, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts, October 1979.

2. OVERVIEW OF THE PADL LANGUAGE

In PADL, a packet communication system is described as a network of *modules.* Each module can either be a *behavior* or *structure* module. It is a behavior module if it is described by a specification of how it processes data and transmits/receives packets. A structure module, on the other hand, is described as a network of modules. Structure modules facilitate the description of systems using structure composition and decomposition techniques. The complete specification of a system comprises a PADL *description,* which is a collection of function definitions and type definitions. A type definition specifies either a new data type or a new module type. A definition for the latter is termed a *module type definition.* A module type specifies the behavior or structure of a class of hardware modules, lists the input and output ports for modules of the class, and gives the data type of packets passing through each port. The system being described is itself a module type. The types of all modules that used as components in the specifications of the system must be defined. For the communication between modules, the type of communication used is packet transmission and is restricted to be single source multiple destinations.

A port of a module type is typed by the type of the data that can be communicated through it. Furthermore a port can either be an input or output port. A class of modules can be specified by parameterizing the module type definition. When a parameterized module type is instantiated, modules belonging to a subset of that class are selected by providing the appropriate parameters. All module types are global in the sense that they can be instantiated as components in structure module type definitions provided that recursive use of the same module type is well formed.

The structure of a structure module is specified as a list of components and a specification of how the components are connected. Each component is a named instantiation of a module type. A connection between components is specified by relating ports. Such a relationship can be specified in two ways – explicitly by specifying the list of the ports that are connected together or implicitly by specifying the ports of other components that are connected to a given component. Conditionals and iterative constructs are provided for facilitating the recursive and iterative specification of structure module types; the latter being particularly useful in specifying highly regular structures. PADL specifications used for connections are termed *connection specifications.*

In the hierarchy of modules of a system, behavior modules are the lowest level modules. The behavior of a module is defined by specifying the operations that are performed with the packets received by the module. Such operations are termed *actions* and include operations for receiving and sending packets, as well as for manipulating state variables. Packets are received from and sent through ports using the **from** and **send** constructs. A packet can be received, nondeterminately, from any one of a group of ports using the **from_either** construct. A packet received using this construct is tagged with the name of the receiving port. The *tagcase* action is provided for selecting expressions based on the port name of the tagged packet.

PADL is a strongly typed language and the data types of all the packets and data used must either be primitive PADL data types or user defined data types. The primitive PADL data types are integers, bit strings (i.e. one dimensional arrays of bits), arrays, records and tagged union types (i.e. data tagged by their types). These can be used to define new data types. Each primitive data type has a predefined set of operations all of which return values. Constructs are provided for constructing complicated expressions. Such constructs include the conditional construct, the **let** construct for associating values with names, and the **tagcase** construct for selecting one of a number of expressions based on the tag of a tagged union value. For manipulating array values, the **forall** construct is provided. All these constructs yield values when evaluated. An expression constructed from these constructs can produce a series of values at each evaluation. However the number of values in the series is fixed for the expression. The number of values yielded is termed the *arity* of the expression.

Frequently used expressions can be parameterized and defined as functions. A function definition is composed of the function name, a list of formal parameters and their types, a list of the types of the returned values, and the expression defining the function. Functions, like expressions, have no side effects and can be used where values are expected.

Functions, expressions, actions, and connection specifications cannot be arbitrarily distributed in a description. As previously stated, functions and expressions can only be used in places where values are expected. Actions can only be used inside module type definitions of behavior modules while connection specifications can only be used in structure module type definitions. Furthermore a module cannot directly access the state variables of another module. Any influence a module has on another can only occur through the explicit transmission of packets between the two modules.

2.1 Notation

In the BNF presentation of the syntax, pairs of large curly braces { ... } indicate zero or more repetitions of the material within. Pairs of large square brackets [...] indicate that the material within may appear at most once.

3. FORMAT OF PADL SPECIFICATIONS

The ASCII character set is used in PADL. The only "control" characters used are the tab and newline characters. The elements in a PADL specification (i.e. the program elements) are operation and punctuation symbols, integer numbers (i.e. a sequence of digits without a decimal point), bit strings, names, subscripted names, and reserved words.

The following are the operation and punctuation symbols:

+	-	*	/	|	&	||
<	>	<=	>=	==	~=	=
:=	:	->	.	;	,	%
(	)	[	]	{	}	~
'	#	@				

Bit strings are sequences of bits. A bit string constant can be expressed as a binary, octal or hexadecimal string. A binary string must be preceded by the single quote character while the # character must precede an octal string. In the case of a hexadecimal string, the string must be preceded by the @ character.

A name is a sequence of letters, digits, or underscores, beginning with a letter. It may be of any reasonable length and may be used as a module name, a module type name, a function name, a port name, a state variable name, a data type name, a value name, a record field name, or a **oneof** tag name. These uses all have their own mechanisms for interpretation, and hence a name may be used without conflict for several of these purposes. For example, a record field name occurs only in a record type specification or record operation, and hence will never be confused with a value name.

There are three forms of subscripted names in PADL. For values and state variables, subscripted names correspond to array names with the subscripts being the array index values. The other two forms of subscripted names are used for ports and components. Like array names, subscript values can be used for these. The list of subscript values come after the name and are delimited by the pair of square brackets [and] for array names, the pair of angle brackets < and > for port names, and the pair of curly braces { and } for component names. Each subscript value occurring in the list of subscript values is separated from each other by commas. Examples of subscripted names are A[15], B<0,5>, C{1, 2, 3}.

There is a set of reserved words that have special meanings in PADL. These words cannot be used as names or identifiers. In this manual these words are printed in **boldface.**

The reserved words are:

abs	**and**	**array**	**at**
begin	**bitstr**	**construct**	**cycle**
do	**else**	**elseif**	**end**
endall	**endcycle**	**endfor**	**endfun**
endif	**endlet**	**endmod**	**endstruct**
endtag	**eval**	**exp**	**external**
false	**for**	**forall**	**from**
from_either	**function**	**if**	**in**
inlet	**integer**	**is**	**let**
make	**max**	**min**	**mod**
module	**nil**	**null**	**oneof**
or	**otherwise**	**outlet**	**plus**
record	**repeat**	**returns**	**rotl**
rotr	**send**	**shifl**	**shifr**
structure	**submodule**	**tag**	**tagcase**
then	**times**	**to**	**true**
type	**until**	**var**	**where**
while			

Upper and lower case letters used in reserved words are not distinguished. However the letters of a name identifier must have consistent capitalization.

The characters separating identifiers, names, reserved words, integer numbers and bit string constants are the space, tab, and newline characters. They may not appear between the individual characters representing one of these program elements. This restriction also applies to those operation symbols that have more than one character. For example in the "not equal" operation symbol ~=, the ~ and = characters cannot be separated by the space, tab, or newline character. In the case of the operation and punctuation symbols which are not alphanumeric characters, the separating characters are not required to separate them from alphanumeric strings.

A comment is delimited by the percent sign character at the beginning and the end of line character at the end. It is is equivalent to a space, and hence may be placed anywhere except within a program element.

Examples of names and constants:

```
ABC3_Q                                  %  A name
34                                      %  An integer constant
'0111100                                %  A binary string
#074                                    %  An octal string
@3C                                     %  A hexadecimal string
```

4. DATA VALUES AND DATA TYPES

The entire collection of values that may be presented to or produced by PADL modules is the value domain of PADL. The value domain is subdivided into distinct disjoint subdomains that are the data types of PADL. There are *basic types* which include the familiar scalar values of computation; *structured types* in the form of arrays and records as defined by the language user in terms of simpler data types; and discriminated union types.

4.1 Data Type Specifications

A data type specification in PADL is a syntactic construct that specifies a data type.

Syntax:

<data type spec> ::= <basic data type spec>
| <compound data type spec>
| <data type name>

<basic data type spec> ::= **null** | **integer** | **bitstr** [[<subscript range>]]

<compound data type spec> ::= **array** [<data type spec> <subscript range>]
| **record** [<field spec> { ; <field spec> }]
| **oneof** [<tag spec> {; <tag spec>}] [**where** <tag def> {, <tag def>}]

<subscript range> ::= <expression> : <expression>

<field spec> ::= <field name> { , <field name> } : <data type spec>

<tag spec> ::= <tag name> { , <tag name> } [: <data type spec>]

<field name> ::= <name>

<tag name> ::= <name>

<tag def> ::= <tag name> { , <tag name> } = <tag value>

<data type name> ::= <name>

<tag value> ::= '<bit string> | #<octal string> | @<hexadecimal string> | <integer number>

<bit string> ::= <binary char> { <binary char> }

<octal string> ::= <octal char> { <octal char> }

<hexadecimal string> ::= <hex char> { <hex char> }

<binary char> ::= ? | <binary digit>

<octal char> ::= ? | <octal digit>

<hex char> ::= ? | <hexadecimal digit>

For the basic data types **null** and **integer**, the specification is simply the name of the type. The data type **bitstr** (i.e. a bit string represented as an one-dimensional array of bits) is specified by giving its name and the optional bounds specification of the index range used to enumerate the bit positions of the bit string. The bounds must be computable at compile time. By convention the leftmost bit is the most significant bit while the rightmost bit is the least significant. All bit string operators are defined with respect to this convention. Its length is the difference between the bounds plus one. When the bounds are specified, they are separated by a colon. If the bounds are identical the bit string has unit length; if they are absent, they are both assumed to be 1. The bound before the colon is the number assigned to the the most significant bit (MSB) while the bound coming after the colon is the number assigned to the least significant bit (LSB). The bounds only affect the bit numbering convention they do not change the convention and internal representation used for bit strings. As an array, **bitstr** is special in the sense that automatic length adjustment takes place in some of its operators.

Examples

```
bitstr          % A single bit
bitstr[0:7]     % A bit string of length 8 with bit 0 as the MSB
bitstr[3:0]     % A bit string of length 4 with bit 3 as the MSB
```

For a compound data type, the specification consists of a *type constructor* giving the name of the compound type followed by the necessary additional information.

The array type constructor gives the type of the elements of the array and the array bounds with the lower bound coming before and the upper bound coming after the colon in the bounds specification. All array bounds have to be computable at compile time, though they do not have to be manifest constants. Unlike bit strings, all array bounds must be specified such that the lower bound is before the colon and the upper bound after it.

Examples:

```
array [ integer 1 : 100 ]                    % A one dimensional array
array [ array [ integer 0 : 15] 0 : 1023 ]   % A two dimensional array
```

The record type constructor gives the field names and the type associated with each field. The field names used within any record specification must be distinct. Where several field names are listed with one type, the fields are all of that type.

Examples:

```
record [ I, J : integer ; FLAG : bitstr ]
record [ I : record [ X : array [ bitstr 1 : 0 ] ; Y : integer ] ;
         TEMP : integer ]
```

A name may be used as a field name and as any other name (but not a reserved word) without conflict, since it is interpreted as a field name only in the record constructor and in record operations. The same field name may be used in several record types without conflict.

The **oneof** (union) type constructor gives the tags and the type associated with each tag. The tag names must be distinct. Where several tag names are listed with one type, the tags all indicate that type. If the colon and following type specification are omitted, the **null** type is assumed. In a **oneof** type specification, the numerical or bit encodings of the tags may optionally be specified using the **where** clause. See 7.3 to see how these encodings are used.

Examples:

```
oneof [ ONE, TWO, FOUR ] where ONE = '001, TWO = '010, FOUR = '100
oneof [ FIX : integer ; BIN : bitstr ]
oneof [ THIS : array [ integer 0 : 9 ] ;
        THAT, THE_OTHER : record [ C : integer ; D : bitstr ]]
```

As in the case of field names, a tag name may coincide with any other name without conflict, and the same tag name may be used in several union types without conflict.

Any type name used as a type specification must be defined by a type definition (see Section 4.5).

4.2 Value Domains

Each data type is a domain of values. Each domain is further characterized by the set of operations that may be used to create and transform values of the type. The operations for each data type of PADL are defined in Chapter 5.

4.3 Basic Types

The Null Type

elements: **nil**

The null type occurs in a distinguished union (**oneof**) type where in one or more alternatives no data value is required.

The Bitstr Type

elements: All bit strings up to some maximum length which is implementation dependent. The bit string ' of length 1 with bounds [1:1] is also denoted by **true**. The bit string '0 of length 1 with bounds [1:1] is also denoted by **false**. All bit strings are stored in binary format.

The Integer Type

elements: The integers between some limits which are implementation dependent.

4.4 Compound Types

Array Types

For each data type defined by some PADL type specification T, an *array type* may be defined by the type specification **array**[T lower-bound : upper-bound]. The given upper bound must be greater than or equal to the given lower bound. If they are equal, the array has exactly one element.

elements: A proper array value in **array**[T lower-bound : upper-bound] consists of a sequence of elements of type T. The number of elements in the sequence is upper-bound − lower-bound + 1.

Record Types

If $T_1,\ldots,T_k$ are PADL type specifications and $N_1,\ldots,N_k$ are distinct names, then **record** [N_1 : T_1 ; . . . ; N_k : T_k] specifies a record type.

elements: Each proper value of the record type is a set of k pairs

$\{(N_1, V_1), \ldots, (N_k, V_k)\}$ where each V_i is an element of T_i.

Union Types

Each element of a union type is an element of one of several constituent types, accompanied by a tag which indicates the constituent type from which the element was taken. If $T_1, \ldots, T_k$ are type specifications, and $N_1, \ldots, N_k$ are distinct names, then **oneof** [N_1 : T_1 ; . . . ; N_k : T_k] specifies a union type.

elements: Each proper element of the union type is a pair (N_i, V_i) where $1 \leq i \leq k$ and V_i is an element of T_i.

4.5 Data Type Definitions

Syntax:

⟨data type def⟩ ::= **type** ⟨data type name⟩ = ⟨data type spec⟩

⟨data type name⟩ ::= ⟨name⟩

A function definition may contain a number of data type definitions which specify user-named data types used in the function. Each data type definition specifies that a data type name denotes the data type represented by the given data type specification. The data type specification part of a data type definition may only contain data type names which have already been defined in other definitions. Recursion and mutual recursion are not permitted in data type definitions. Such data type definitions may be used to construct data types composed of array or record structures of unlimited depth.

A name may be used as a data type name and as any other kind of name without conflict, since it is interpreted as a data type name only in well defined contexts i.e. whenever a data type specification is permitted.

5. OPERATIONS

In this section we specify the sets of operations applicable to each data type of PADL. In the examples of notation, M and N stand for bit strings, I, J, K and L for integers, A and B for arrays, R for records, U for union (**oneof**) values, T for arbitrary types, and V for values of arbitrary type.

5.1 Bit String Operations

A rich set of operations is available in PADL for bit strings. For these operators, the least significant bit is the rightmost bit while the leftmost bit is the most significant.

operation	notation	functionality
and	M & N	bitstr, bitstr → bitstr
or	M | N	bitstr, bitstr → bitstr
not	~ M	bitstr → bitstr
equal	M = = N	bitstr, bitstr → bitstr
not equal	M ~= N	bitstr, bitstr → bitstr
concatenate	M || N	bitstr, bitstr → bitstr
shift left	shifl(M, J)	bitstr, integer → bitstr
shift right	shifr(M, J)	bitstr, integer → bitstr
rotate left	rotl(M, J)	bitstr, integer → bitstr
rotate right	rotr(M, J)	bitstr, integer → bitstr

and M & N

The shorter argument is left-extended with 0's to match the length of the longer argument. The two bit strings are then ANDed bitwise to obtain a bit string of the same length as the longer argument.

or M | N

The shorter argument is left-extended with 0's to match the length of the longer argument. The two bit strings are then ORed bitwise to obtain a bit string of the same length as the longer argument.

not ~ M

The argument is bitwise complemented to obtain a bit string of the same length.

equal M = = N

The shorter argument is left-extended with 0's to match the length of the longer argument. The two bit strings are then compared. The result is **true** if the two bit strings match at every bit position, **false** otherwise.

not equal M ~= N

The shorter argument is left-extended with 0's to match the length of the longer argument. The two bit strings are then compared. The result is **false** if the two bit strings match at every bit position, **true** otherwise.

concatenate M || N

Let the lengths of M and N be m and n, respectively. M || N returns a bit string whose length is m+n, formed by concatenating M and N. The substring of M || N consisting of its rightmost n bits is identical to N while the substring of M || N consisting of its leftmost m bits is identical to M.

shift left **shifl(M, J)**

The bits of the string M are shifted to the left by J positions. The leftmost J bits of M are truncated. The rightmost J bit positions of M are filled with zeroes.

shift right **shifr(M, J)**

The bits of the string M are shifted to the right by J positions. The rightmost J bits of M are truncated. The leftmost J bit positions of M are filled with zeroes.

rotate left **rotl(M, J)**

The bits of the string M are rotated to the left by J positions.

rotate right **rotr(M, J)**

The bits of the string M are rotated to the right by J positions.

The operations on bit strings also include selection of substrings. For convenience an abbreviated notation is included for selecting substrings of length 1.

operation	notation	functionality
select substring of length 1	M[J]	bitstr, integer → bitstr
select substring	M[J:K]	bitstr, integer, integer → bitstr

The integer division operation computes the integer quotient of its arguments while the integer modulus operation returns the integer remainder of the division of its arguments. In either case, the second argument is the divisor.

5.3 Array Operations

The operations for the array data type **array**[T lower-bound : upper-bound] include element selection and subarray selection of a given array. Recall that an array value consists of a range defined by a low index LO (i.e. the lower bound), a high index HI (i.e. the higher bound), and a sequence of HI-LO+1 elements of the given type.

operation	notation	functionality
select element	A[J]	array[T], integer → T
select subarray	A[J:K]	array[T], integer, integer → array[T]

Select element A[J]

This operation yields the element of the array A at index J. J must be within the range of the array, i.e. $HI \geq J \geq LO$.

Select subarray A[J:K]

This operation yields the array whose elements are those of the array A at indices J through K. J and K must be expressions with integer values within the range of the array, i.e. $HI \geq J \geq K \geq LO$.

5.4 Record Operations

The operations for a record type specified as T = **record**[$N_1 : T_1 ; \ldots ; N_k : T_k$] where $N_1 \ldots N_k$ are the field names, and $T_1 \ldots T_k$ are the corresponding types, are the following.

operation	notation	functionality
create	**record**[$N_1 : V_1 ; \ldots ; N_k : V_k$]	$T_1, \ldots, T_k \rightarrow T$
$select_i$, $1 \leq i \leq k$	$R . N_i$	$T \rightarrow T_i$

Create **record**[$N_1 : V_1 ; \ldots ; N_k : V_k$]

This builds a record value { $(N_1, V_1), \ldots, (N_k, V_k)$ }. All of the field names associated with the type of the record being constructed must appear in the list. This operation is useful for initializing records.

Select R . N

This returns the value of the named field, that is, V_i if $N = N_i$.

5.5 Operations for union types

The basic operations for a union type specified as $T =$ **oneof** [$N_1 : T_1 ; \ldots ; N_k : T_k$] are a create operation and a test of a tag. The **tagcase** control structure explained in Section 7.3 is the mechanism for accessing constituent values from a value of union type. In the following, $N_1 \ldots N_k$ are the tag names, and $T_1 \ldots T_k$ are the corresponding constituent types.

operation	notation	functionality
create$_i$, $1 \leq i \leq k$	**make** T [$N_i : V$]	$T_i \rightarrow T$
tag test$_i$, $1 \leq i \leq k$	**is** N_i (U)	$T \rightarrow$ bitstr

The operations **make** T [N : V] and **is** N (U) are type-correct only if N is a tag name of the type T and V is of that constituent type. The result of **make** T [N_i : V] is the pair (N_i : V) for any element V of T_i. The result of **is** N_i (U) is **true** if U = (N_i, anything), otherwise it is **false**.

5.6 Type Conversion Operators

Conversion operators are provided for converting between integers and bit strings.

operation	notation	functionality
bit string to integer	**integer**(M)	bitstr → integer
integer to bit string	**bitstr**(J, K)	integer, integer → bitstr

bit string to integer **integer**(M)

The bit string M is considered as the binary representation of a *non-negative* integer and converted

accordingly. If the length of the bit string M exceeds that used for implementing integers, M is left truncated before the conversion is performed. Hence if integers are implemented as bit strings of length K and M is of length L, then the leftmost K-L bits of M are dropped and the resulting truncated bit string of length L is converted to an integer.

integer to bit string **bitstr(J, K)**

J is the argument to be converted into a bit string K bits long, and must be a *non-negative* integer. The binary representation of J is left-extended with 0's or left-truncated as necessary to form a bit string of length K.

5.7 Type Correctness of Operations

In PADL the type of value produced by each expression can be determined by the translator from the properties of the operations as specified in this section. An operation in a PADL description is type correct if and only if the types of its argument expressions are the same as the argument types specified for the operation. Note that for each operator the types of the results are determined when the types of the arguments are known.

6. CONSTANTS, VALUE NAMES, AND EXPRESSIONS

An expression is the basic syntactic unit denoting a tuple of values of some types. The *arity* of an expression is the size of the tuple of values it denotes. Two expressions are said to *conform* if they have the same arity and the corresponding values are of the same type. The design of the PADL language is such that the arity and types of an expression, and hence the conformity of two expressions, may be determined by inspection of the program. The simplest type of expression of arity one is a constant, a value name, or an operation applied to other expressions of arity one. The simplest type of expression of higher arity is a series of expressions of arity one separated by commas.

6.1 Constants

A constant is a syntactic unit of arity one whose value and type are manifest from its form.

Syntax:

<constant> ::= nil | true | false | <integer number> | <bit string constant>
<bit string constant> ::= '<bit string> | #<octal string> | @<hexadecimal string>
<bit string> ::= <binary char> { <binary char> }
<octal string> ::= <octal char> { <octal char> }
<hexadecimal string> ::= <hex char> { <hex char> }
<binary char> ::= ? | <binary digit>
<octal char> ::= <binary char> | <octal digit>
<hex char> ::= <octal char> | <hexadecimal digit>

The only constant of the **null** type is the reserved word **nil.**

The reserved words **true** and **false** denote the bit strings '0 and '1 respectively.

The only constants of the **integer** type are non-negative **integers.**

A binary bit string constant is a bit string preceded by a single quote, e.g., '001100. A bit string constant may also contain the special "don't-care" symbol "?", e.g. '00??11. A bit string constant containing one or more don't-care symbols denotes the set of bit strings which matches it at all positions not marked by "?". Such a bit string constant may only be used for identifying an arm in a **tagcase** construct.

A bit string constant can be denoted as an octal string, which consists of octal digits preceded by the "#" character, or a hexadecimal string consisting of hexadecimal digits preceded by the "@" character. The letters in the hexadecimal string can be upper or lower case. For example, #1777 and @3FF (or @3ff) both denote the binary string '001111111111. The "don't-care" symbol "?" can also be used. Hence #0?67 is equivalent to '000???110111 while @0?A is equivalent to '0000????1001.

There are no array, record, or union constants, but various constructing operators may be used with constant arguments to denote "constant" arrays, records, or union elements.

Examples:

```
record [ A : 6 ; B : true ]          (record constant)
make T [ A : 6 ]                     (constant of union type T)
```

6.2 Value names

A value name is a name which denotes either a value received or sent through a module port, or a single computed value of a given type. Every value name is introduced either in the header of a module type definition (if the value name is a formal parameter or a port of the module being defined), in state variable declarations, in the header of a function definition (if the value name is a formal argument of the function being defined) or in a program construct such as a **let** block or a **forall** block. In every case, each value name has a *scope* and a type. The scope of a value name is the region of program text in which a reference to the value name denotes its value. The scope and type of any value name may be determined by inspection of the program construct that introduces it.

The scope of a value name introduced as a parameter or module port is the entire module type definition, less any inner scopes that re-introduce the same value name. The type of such a value name is given by a type declaration in the module header (see Section 10.2).

Example:

```
type M = module (inlet INPUT : integer; outlet OUTPUT : bitstr)
   <rest of module definition>
endmod
```

State variable declarations may be given only in module type definitions in which module behavior is specified. Syntactically they are placed immediately after a module header and have the same scope as the port names introduced in that header.

Example:

```
var V : integer (state variable declaration, see Section 10.2.1)
```

The scope of a value name introduced as a formal argument of a function is the entire function definition, less any inner scopes that re-introduce the same value name. The type of such a value name is given by a type declaration in the function header. Its value is the value of the corresponding argument for the relevant invocation of the function. In the following example, an appearance of value name X in the expression denotes the value of the argument with which F was invoked. Its type is integer.

Example:

```
function F ( X : integer returns bitstr )
   <expression>
endfun
```

The scope of a value name introduced in a program construct such as a let or forall block is some region of the construct that depends on the nature of the construct, less any inner scopes that re-introduce the same value name. The manner in which the type and value of the value name are established depends on the form of the construct.

Example:

```
let
   X : integer = 3 ;
   <another declaration or definition> ;
   <another declaration or definition> ;
   <another declaration or definition> ;
in <expression>
endlet
```

The scope of X is the entire block, including the expression after in, less any inner scopes that re-introduce X. Its type is integer; its value is 3. The let construct is described in Section 7.2. If this block had appeared within the scope of X introduced by some outer construct, that outer scope, with its value and type, would disappear within this let block.

6.3 Expressions

Expressions are built out of smaller expressions by means of operation symbols.

Syntax:

<expression> : : = <level1 exp> | <expression> , <level1 exp> (the arities are added)

In the next eight lines, the operator may only be used if all operands are of arity one.

Syntax:

<level1 exp> : : = <level2 exp> | <level1 exp> | <level2 exp> (bit string "or")
<level2 exp> : : = <level3 exp> | <level2 exp> & <level3 exp> (bit string "and")
<level3 exp> : : = <level4 exp> | ~ <level4 exp> (bit string "not")
<level4 exp> : : = <level5 exp> | <level4 exp> <relational op> <level5 exp>
<level5 exp> : : = <level6 exp> | <level5 exp> || <level6 exp> (bit string concatenate)
<level6 exp> : : = <level7 exp> | <level6 exp> <adding op> <level7 exp>
<level7 exp> : : = <level8 exp> | <level7 exp> <multiplying op> <level8 exp>
<level8 exp> : : = <primary> | <unary op> <primary>

<relational op> : : = < | <= | > | >= | == | ~=
<adding op> : : = + | -
<multiplying op> : : = * | /
<unary op> : : = + | -

<primary> : : = <constant> | <value name> (these have arity one)
| (<expression>) (same arity as expression in parentheses)
| <function invocation> (arity is the number of values returned)
| <array ref> | <array generator> (these have arity one)
| <record ref> | <record generator> (these have arity one)
| <oneof test> | <oneof generator> (these have arity one)
| <prefix operation> (this has arity one)
| <conditional exp> (the following four constructs
| <letin exp> are discussed Chapter 7)
| <tagcase exp>
| <forall exp>

⟨value name⟩ : : = ⟨name⟩

In a function invocation, the arity of the expression in parentheses must be equal to the number of arguments required by the function. For an array reference, the arity of the expression used in the subscripts must be equal to the dimension of the array.

Syntax:

⟨function invocation⟩ : : = ⟨function name⟩ (⟨expression⟩)

⟨function name⟩ : : = ⟨name⟩

⟨array ref⟩ : : = ⟨primary⟩ [⟨subscripts⟩]

⟨record ref⟩ : : = ⟨primary⟩ . ⟨field name⟩

⟨subscripts⟩ : : = ⟨expression⟩

In the next 8 forms, all expressions must have arity one and the resultant expressions always have arity one.

Syntax:

⟨array generator⟩ : : = ⟨primary⟩ [⟨subscript range list⟩]

⟨subscript range list⟩ : : = ⟨subscript range⟩ { , ⟨subscript range⟩ }

⟨subscript range⟩ : : = ⟨expression⟩ : ⟨expression⟩

⟨record generator⟩ : : = **record** [⟨field name⟩ : ⟨expression⟩ { ; ⟨field name⟩ : ⟨expression⟩ }]

⟨field name⟩ : : = ⟨name⟩

⟨oneof test⟩ : : = **is** ⟨tag name⟩ (⟨expression⟩)

⟨oneof generator⟩ : : = **make** ⟨data type spec⟩ [⟨tag name⟩ : ⟨expression⟩]

⟨tag name⟩ : : = ⟨name⟩

The arities of the argument expressions for a prefix operation are as shown, and the result arity is always one.

Syntax:

```
<prefix operation> ::= abs (<expression>)          (arity = 1)
                     | exp (<expression>)          (arity = 2)
                     | mod (<expression>)          (arity = 2)
                     | shifl (<expression>)        (arity = 2)
                     | shifr (<expression>)        (arity = 2)
                     | rotl (<expression>)         (arity = 2)
                     | rotr (<expression>)         (arity = 2)
                     | bitstr (<expression>)       (arity = 2)
                     | integer (<expression>)      (arity = 1)
```

Note that operators obey the customary precedence rules: unary plus and minus have highest priority; multiplicative operators (*, /) are next; additive operators (+, -) are next; "||" is next; relational operators (<, <=, >, >=, =, ~=) are next; "~" is next; "&" is next; and "|" has the lowest priority.

Examples of expressions of arity one:

```
A
true
'001100
'001 || A
- X + 3 * B
3 * ( X + Y )
func(3+X, Y)                    (if "func" returns one value)
A [ 3 : Z ]
A [ 4, J ]
R . X . Y . ZZ
array [ 10 : 2 ]
record [ A : P ; B : Q ]
is A (U)
make T [A : 3]
if P then 4 else 5 endif        (see Chapter 7)
rotr(M,3)
bitstr(15,4)
```

6.4 Array Operations for Multi-dimensional Arrays

Elements of multi-dimensional arrays are selected by giving the name and array indices for the element as an expression. Hence in:

```
A[J][K][L]
```

the selected element of the three-dimensional array A has array index values of J, K, and L for the first, second, and third dimensions. These array indices are represented by the expression (of arity three) within the square brackets. The above can be abbreviated to:

```
A[J, K, L]
```

6.5 Expressions of Higher Arity

The program structures provided in PADL for conditional computation and iteration are expressions of arbitrary arity, and are described in Chapter 7. Such expressions, or function invocations, may occur in program text in places that require a tuple of values of specified types: the argument list of an operation or function invocation, the body of a function definition, a list of array indices or elements in an array operation, or in building the program structures presented in Chapter 7.

6.6 Function Invocations

A function invocation consists of the name of the function followed by an argument list within parentheses. (The syntax is the same for internal and external functions.) The argument list is an expression, whose arity and types conform to the arguments required by the function. This information is given in the header of the function definition (see Section 10.1). The argument list is usually written as a series of expressions of arity one separated by commas, but it may be any expression.

A function invocation is itself an expression whose arity and types are the number and types of the values returned by the function. Information on the number and types of the returned values also appears in the function's header. An invocation that returns one value may appear in expressions with complete generality, such as an argument to arithmetic, array, and record operations. An invocation that returns several values may only be used where expressions of higher arity are permitted.

In the following examples, SINGLE, DOUBLE, and TRIPLE each take 3 arguments and return 1, 2, or 3 values, respectively:

```
K = 3 + Z * SINGLE (X + 1, 3, SINGLE (X + 2, 4, W)) ;
```

In the following example, if P is **false**, F and G are defined to be DOUBLE (X, Y, Z), while H is defined to be W:

```
F, G, H = if P then TRIPLE (X, Y, Z) else DOUBLE (X, Y, Z), W endif ;
```

Since the argument list for any function may be any expression, it may be a multiple-result function invocation or other program structure.

```
3 + SINGLE (TRIPLE (X, Y, Z))

3 + SINGLE (P, DOUBLE (X, Y, Z))

4 + SINGLE (if P then 4, 5 else DOUBLE (P, Q, R) endif, X)
```

The last example invokes SINGLE with three arguments, of which the first two are either 4 and 5 or the two values returned by DOUBLE. The third argument to SINGLE is always X.

If an actual parameter in a function invocation is a bit string of length m, the formal parameter is declared to be a bit string type of length n, and m is not equal to n, then the actual parameter is left-extended with 0's or left truncated to form a bit string argument of length n for the function invocation.

7. STRUCTURED EXPRESSIONS

The program structures described in this section are specific forms of expressions. If their arity is one, they may appear in arithmetic operations.

Example:

```
if P then X else Y endif + 3
```

This expression has value X+3 or Y+3, depending on P.

7.1 The IF Construct

The conditional expression selects one of several expressions, depending on which conditions are satisfied.

Syntax:

<conditional exp> : : = **if** <condition> **then** <expression>
{ **elseif** <condition> **then** <expression> }
else <expression>
endif

<condition> : : = <expression>

The conditions following **if** and **elseif** are *test expressions*. Their arity must be one and their type **bitstr of** length 1. The expressions following **then** and **else** are the *arms*. They must conform to each other, and **the** entire construct conforms to **the arms.**

The entire construct is an expression whose tuple of values is that of the first arm whose test expression is true, or the final arm if all test expressions are false.

The **if** construct introduces no value names. All value name scopes pass into an **if** construct. If **the** scope of a value name includes an **if** construct, it includes all of the expressions of that construct, so that **value** name may be used anywhere inside the conditional construct.

7.2 The LET Construct

The purpose of this construct is to introduce one or more value names, define their values, and evaluate an expression within their scope (that is, making use of their defined values).

Syntax:

⟨letin exp⟩ ::= **let** ⟨decldef part⟩
in ⟨expression⟩
endlet

⟨decldef part⟩ ::= ⟨decldef⟩ { ; ⟨decldef⟩ } [;]

⟨decldef⟩ ::= ⟨decl⟩
| ⟨def⟩
| ⟨decl⟩ { , ⟨decl⟩ } = ⟨expression⟩

⟨def⟩ ::= ⟨name⟩ { , ⟨name⟩ } = ⟨expression⟩

⟨decl⟩ ::= ⟨name⟩ { , ⟨name⟩ } : ⟨data type spec⟩

Every value name introduced in a **let** block must be declared exactly once and defined exactly once in that block. The declaration may be part of the definition, or it may be by itself preceding the definition.

Examples:

```
X : integer ;                    % Declaration
X = 3 ;                          % Definition
Y : integer = 4 + Q ;            % Declaration as part of definition
```

Each value name must be defined before it is used (on the right hand side of another definition). Declarations and definitions may be mixed in any order as long as these requirements are met.

Several value names may be declared at once:

```
X, Y, Z : integer ;
```

This declares all 3 names to be **integer.**

Several value names may be defined at once. The number and types of the names must conform to the arity and types of the expression on the right hand side.

```
X, Y, Z = 1, 2, 3 ;
P, Q, R = TRIPLE(X, Y, Z) ;
```

Several value names may be declared and defined at once. In this case, each of a group of value name names preceding a type specification are declared to be of that type.

```
X : integer, Y, Z : bitstr = 3, true, false ;
```

This declares X to be integer, and both Y and Z to be bit strings of length 1.

The declarations, definitions, and combined declarations and definitions are separated by semicolons.

The scope of each value name introduced in a **let** block is the entire block less any inner constructs that re-introduce the same value name. However, a value name must not be used in the definitions preceding its own definition.

All scopes for value names not introduced in a given **let** block pass into that block. Hence, if the scope of a value name (introduced by an outer construct) includes a **let** block and that value name is not re-introduced, it may be referred to freely within the block.

Example:

```
let X , T : integer ;
    T = P + 37 ;
    X = T + 24 ;
in X * T
endlet
```

In this example, the value of P is imported from the outer context. The scopes of T and X are both the entire block. A reference to X in the definition of T would be illegal because it is within the scope of X but does not follow the definition of X. The arity of this construct is one, and its type is integer, because X*T has arity one and type **integer**.

Since a value name may not be used until after it has been defined, and must be defined only once in a block, it may not appear in its own definition. Hence definitions such as

```
I = I + 1 ;
```

are never legal in **let** blocks.

In a definition, if the value name is declared to be a bit string of length m and its defining value evaluates to a bit string of length n, the defining value is left-extended with 0's or left-truncated, as necessary, to form a bit string of length m for use in the definition.

Example

```
let M : bitstr[0:3]; N : bitstr[0:15] = @F6
in
    integer(M), rotr(N,4)
endlet
```

The above let construct returns two values – the integer value 6 (i.e. the value of the truncated bit string @6) and the bit string (its length = 16) of value @600F

The expression following the reserved word in is in the scope of all of the introduced value names, and hence can make use of their definitions. The entire let construct conforms to this expression.

7.3 The TAGCASE Expression

This selects one of a number of expressions, depending on the tag of a oneof value, and extracts the constituent value.

Syntax:

⟨tagcase exp⟩ ::= **tagcase** [⟨value name⟩ =] ⟨expression⟩ [;]
⟨tag list⟩ : ⟨expression⟩ [;]
{ ⟨tag list⟩ : ⟨expression⟩ [;] }
[**otherwise** : ⟨expression⟩ [;]]
endtag

⟨tag list⟩ ::= **tag** ⟨tag⟩ { , ⟨tag⟩ }

⟨tag⟩ ::= ⟨tag value⟩ | ⟨name⟩

⟨tag value⟩ ::= ⟨bit string constant⟩ | ⟨integer number⟩

The entire construct is an expression whose values are those of the expression in the arm whose tag name or tag value matches that of the value of the test expression. If no match is found, the arm following the reserved word otherwise is used. All arms must conform to each other, and the entire construct conforms to the arms.

The expression following the reserved word **tagcase** must be of arity one and of a **oneof** type. The tag names appearing in the arms of the construct must be tags (i.e. tag names or tag values) of that **oneof** type. If they comprise all the tags of that type, the **otherwise** arm is not used; if not, the **otherwise** arm is required.

If a value name and "=" appear after the reserved word **tagcase**, that name is introduced for each arm of the construct except the **otherwise** arm. Its scope in each case is the expression in that arm, and its type is the constituent type indicated by the tag name for that arm. If an arm is evaluated (meaning that the tag of the test expression matches the tag name of the arm), the value name is defined to be the constituent value from the test expression. If the value name and "=" do not appear, the constituent value is not made available inside the arms.

Example:

Let X be of type

```
oneof [ A : integer ; B : array[integer 1 : 10] ; C : integer ; D : bitstr ]
```

If X has tag A and constituent value 3,

```
tagcase P = X ;
    tag A : P + 4
    tag B : P[5]
    otherwise : 5
endtag
```

has value 7. The first arm is taken, and P (whose type is **integer** in that arm) is defined to be 3, the constituent value of X. If X has tag B and constituent value some array whose fifth element is 2, the value of the above construct is 2. In that case, P is defined to be the array. If X has tag C or D, the construct has value 5. In that case the constituent value is not made available, since the value name's scopes do not include the **otherwise** arm. (This is because the **otherwise** arm can encompass different constituent types, so the type of the value name could not be determined.)

More than one tag name may share the same arm if they indicate the same type. In this case, the tag names are all listed, separated by commas, after the reserved word **tag**.

Example:

Let X be of type

```
oneof [ A : integer ; B : bitstr ; C : integer ]
```

Then the following is permissible:

```
tagcase P = X ;
   tag A, C : <expression1>          % P is an integer here
   tag B : <expression2>             % P is a bit here
endtag
```

All scopes of value names other than the one appearing after the reserved word tagcase pass into the construct. An outer scope for a value name with the same name as the one appearing after the reserved word tagcase does not pass into the tagcase construct.

For the tags, bit or numerical encodings instead of the tag names can be used. Let A be of type

```
oneof [ ONE, TWO, FOUR : integer ] where ONE = '001, TWO = '010,
                                         FOUR = '100
```

Hence the following tagcase construct can be used.

```
tagcase V = A
       tag '??0 : <expression1>      % Even valued tags
       otherwise : <expression2>     % Odd valued tags
endtag
```

The above tagcase construct uses bit string constants as tags for specifying the expressions for even (i.e. tag names TWO, FOUR) and odd valued bit strings (i.e. tag name ONE). It can be rewritten as:

```
tagcase V = A
       tag TWO, FOUR : <expression1>
       tag ONE : <expression2>
endtag
```

Since at most one arm of the tagcase construct can be entered at any one time, tag values used as tags in a tagcase construct must be distinct if they are in different arms of the construct. Hence when bit string constants as are used tags, they must not overlapped. For example the tags '0?0 and '??0 cannot appear on different arms of the tagcase construct.

7.4 The FORALL Construct

This generates one or more sets of values, of uniform type within each set, and either returns them as arrays or returns the result of some operation (such as addition) on them. The former case is indicated by the reserved word **construct**, the latter by the word **eval** followed by the name of the operator.

This construct introduces one or more *index* value names of type integer and a number of optional temporary value names, the latter in the the same manner as in a **let** block.

Syntax:

<forall exp> : : = **forall** <value name> **in** [<expression>] { , <value name> **in** [<expression>] }
[<decldef part>]
<forall body part>
{ <forall body part> }
endall

<forall body part> : : = **construct** <expression> | **eval** <forall op> <expression>

<forall op> : : = **plus** | **times** | **min** | **max** | **or** | **and**

The index names are those appearing before the reserved word **in**. The temporary names are those appearing in the declarations and definitions. Index and temporary names must all be different. Their scopes are the entire construct less any inner blocks that re-introduce the same value name. The types of the indices are integer. The types of the temporary names are specified in their declarations. As in a **let** expression, a temporary name may not appear in definitions preceding its own.

Each expression appearing in square brackets after the reserved word **in** is of arity two with both components of type **integer**. The two components are the low and high limits, inclusive and appearing in that order, for the index. For each number within those limits, the index is defined to be that number, the definitions of the temporary names are made, and all the parts are evaluated. When more than one index is given, this is done for each point in the "Cartesian product" of the ranges, that is, for every combination of index values.

In a **construct** part, the expression is evaluated for each index value, and for each component of the expression, an array is formed having the same limits as the limits given for the index and elements equal to the values obtained. If more than one index is given, a multidimensional array is formed, that is, an array of arrays, with the first index referring to the outermost array. The following expression,

```
forall J in [ 1, 4 ]
X : bitstr[0:1] = bitstr(J-1,2);
construct J, X, J mod 3;
endall
```

creates 3 arrays, all with index range 1 to 4. The first is an integer array with element having values 1, 2, 3, and 4. The second is an array of bit strings each of length 2 and the array contains '00, '01, '10, and '11. The last is an integer array with elements 1, 2, 0, and 1. This **forall** block is an expression of arity three whose values are these three arrays.

The following **forall** expression,

```
forall J in [ A, B ], K in [ C, D ]
construct <expression>
endall
```

is equivalent to

```
forall J in [ A, B ]
construct
        forall K in [ C, D ]
        construct <expression>
        endall
endall
```

and constructs a two-dimensional array, that is, an array whose limits are [A, B] and whose elements are arrays whose limits are [C, D].

In an **eval** part, the operation must be one of **plus**, **times**, **min**, **max**, **or**, or **and**. The arity of the expression must be one, and its type must be appropriate for the operation: integer for **plus**, **times**, **min**, or **max**, **bitstr** for **or** or **and**. The expression is evaluated for each index value, and the operation is performed on the collection of values that are produced. If multiple indices are used, the operation is performed on the entire collection of values produced for all combinations of index values. For example the expression,

```
forall J in [ 1, N ]
eval plus J*J
endall
```

returns $\sum_{j=1}^{N} j^2$.

The result of an entire **forall** block is an expression constructed by concatenating the results of all of the parts. Hence the expression,

```
forall J in [ 1, 10 ]
X : bitstr = bitstr(J mod 2, 1) ;
eval plus J*J
construct J, X
endall
```

is an expression of arity 3 and types **integer**, **array[integer** 1 : 10], and **array[bitstr** 1 : 10].

The scopes of any value names other than the index and temporary names, introduced in outer constructs, pass into the **forall** block.

8. ACTIONS

Actions in PADL are like imperative statements in programming languages. Actions specify events that take place in a hardware module, changing the values (state information) held in one or more state variables in the module, receiving an input packet from an input port, or sending an output packet at an output port. In some of these actions, values are produced. Actions can only appear inside a behavior module type definition.

8.1 Elementary Actions

The elementary actions in PADL are updating state variables, receiving input packets, and transmitting output packets. In these actions an expression of some type is evaluated and its value is then associated with a typed identifier. State variables are unlike value names in that only state variables can be updated and have to be explicitly declared. As in let definitions and function invocations, only implicit conversions between bit strings of different lengths is supported in PADL. Let an expression evaluate to a bit string of length m and let the target identifier (state variables in assignments, local identifiers in let definitions, output ports in output actions, actual parameters in function invocations) be declared to be a bit string type with length n. If m is not equal to n, the result of the expression evaluation is left-extended with 0's (if m < n) or left truncated (if m > n) to form a bit string of length n before the action continues.

Syntax:

⟨elementary action⟩ ::= ⟨state variable assignment⟩
| ⟨input action⟩ | ⟨output action⟩

State variable assignment

Syntax:

⟨state variable assignment⟩ ::= ⟨state var⟩ { , ⟨state var⟩ } := ⟨actval⟩
⟨actval⟩ ::= ⟨expression⟩ | ⟨input actions⟩
⟨state var⟩ ::= ⟨name⟩ | ⟨state var array ref⟩ | ⟨state var record ref⟩
⟨state var array ref⟩ ::= ⟨state var⟩ [⟨subscript range list⟩]
⟨subscript range list⟩ ::= ⟨subscript range⟩ { , ⟨subscript range⟩ }
⟨subscript range⟩ ::= ⟨expression⟩ : ⟨expression⟩
⟨state var record ref⟩ ::= ⟨state var⟩ . ⟨field name⟩

<field name> : : = <name>

In a state variable assignment, one or more state variables are assigned values from an expression. The state variables may be basic or structured data types. Each state variable assignment must fall within the lexical scope in a module type definition in which the state variable has been declared.

Input action

Syntax:

<input action> : : = **from** <port id list> | <tagged from>
<tagged from> : : = **tagcase** [<value name> =] <from-either list> [;]
<tag list> : <expression> [;]
{ <tag list> : <expression> [;] }
[**otherwise** : <expression> [;]]
endtag
<from-either list> : : = **from_either** <port id> , <port id list>
<tag list> : : = **tag** <port id list>
<port id list> : : = <port id> { , <port id> }
<port id> : : = <name> { < <subscripts> > }
<subscripts> : : = <expression>

A **from** operation returns a value from each input port listed. The returned value is of the type associated with the named input port in the enclosing module header. The **from** operation is completed only when values are received from all the ports listed. A **from_either** operation returns a value received from one of the named input ports, tagged by the name of the input port from which this value is received. Thus the value returned by the operation

from_either port1, ..., portn

belongs to the data type

oneof [port1 : data-type-1, ..., portn : data-type-n]

where *data-type-i* is the data type associated with *porti* in the enclosing module header definition.

If a value name and "=" come before the reserved word **tagcase**, that name can be used in each arm of the construct in a manner similar to the **tagcase** construct described in Section 7.3. The *port id*(s) given in the tag list is used for selecting an arm of the construct. The type of the value name must be the same as the packet type of the port from which the value is received.

Example:

```
INDIR , OUTDIR : bitstr,  X: bitstr[0 : 8] =
tagcase Y = from_either IN0, IN1
    tag IN0 : '0, Y[0], Y;
    tag IN1 : '1, Y[0], Y;
endtag
```

In the above example, assume that the input ports IN0 and IN1 have been previously declared to be of type **bitstr**. The arity of the expression in each arm is 3. The value of INDIR is '0 if the packet is from port IN0 and '1 if it is from port IN1. X and OUTDIR have the values of the packet received and bit 0 of the packet, respectively.

The **from_either** operation is the only source of nondeterminacy in a PADL behavior description.

Output action

Syntax:

<output action> ::= **send** <expression> **at** <port id list>

A **send** action transmits a list of packets, at the named output ports. The values of the packets are denoted by *expression*. A **send** action is completed only after each packet sent is received at the receiving input port(s). The correspondence between output and input actions in intermodule communication is explained in Section 8.3.

8.2 Compound Actions

Syntax:

<compound action> ::= <elementary action>
| <action block>
| <conditional action>
| <tagcase action>

| <iteration>
| <definition block>

action-block

Syntax:

<action block> : : = **begin** <compound action> { ; <compound action> } **end**

An action block consists of a sequence of actions delimited by the reserved words **begin** and **end.**

conditional-action

Syntax:

<conditional action> : : = **if** <condition> **then** <compound action>
{ **elseif** <condition> **then** <compound action> }
[**else** <compound action>]
endif
<condition> : : = <expression>

The condition expression must be of arity one and type **bitstr** of length 1. If this expression evaluates to **true**, the **then** branch is entered and executed. If this expression evaluates to **false** and at least one **elseif** or **else** branch is present, the first **elseif** or **else** branch is entered and executed.

tagcase-action

Syntax:

<tagcase action> : : = **tagcase** [<value name> =] <expression> [;]
<tag list> : <compound action> [;]
{ <tag list> : <compound action> [;] }
[**otherwise** : <compound action> [;]]
endtag
<tag list> : : = **tag** <tag> { , <tag> }

⟨tag⟩ : : = ⟨tag value⟩ | ⟨name⟩

⟨tag value⟩ : : = ⟨bit string constant⟩ | ⟨integer number⟩

The *tagcase-action* construct has the same syntax and the same flow of control in its evaluation as the *tagcase-expression* construct explained in Section 7.3, except that each arm consists of actions instead of expressions.

iteration

Syntax:

⟨iteration⟩ : : = **while** ⟨condition⟩ **do** ⟨compound action⟩
| **repeat** ⟨compound action⟩ **until** ⟨condition⟩

In executing a **while** construct, the sequence of actions specified in *compound-action* is executed once every time the specified condition is satisfied (i.e., evaluates to **true**). Execution of the **while** construct terminates as soon as the condition evaluates to **false**.

In executing a **repeat** construct, the sequence of actions specified in *compound-action* is executed first, and then reexecuted once every time the specified condition is not satisfied. Execution of the **repeat** construct also terminates as soon as the condition evaluates to **true**.

definition-block

The purpose of this construct is to introduce one or more value names, define their values, and perform a group of actions within their scope (that is, making use of their defined values). Note that the values introduced can be those obtained through input operations. This is the only way of accessing values from input operations without involving a state variable.

Syntax:

⟨definition block⟩ : : = **let** ⟨actdecldef part⟩
in ⟨compound action⟩
endlet

<actdecldef part> ::= <actdecldef> { ; <actdecldef> } [;]

<actdecldef> ::= <decl>
| <def>
| <decl> { , <decl> } = <actval>

Scope rules for identifiers introduced in *definition-blocks* are identical to those for identifiers introduced in let expressions (Section 7.2).

8.3 Intermodule Packet Communication

Modules in PADL communicate by sending packets to each other. Packets are transmitted between modules over channels. A channel connects an output port of the sending module to an input port of the receiving module. Channel connections between submodules are specified in structure module type definitions. The semantics of input and output actions of PADL is described below.

A module receives a packet from a channel by performing an input action on the input port connected to that channel. A packet is transmitted on a channel by performing an output action on the output port connected to that channel. Coordination between input and output actions performed on the ports connected to a channel are defined in terms of channel state transitions. A channel can be in one of two states: empty (its initial state), or full. An output action on an output port connected to a channel may proceed only if the channel is empty. An empty channel becomes full after an output action is performed on the output port connected to it. An input action on an input port connected to a channel may proceed only if the channel is full. A full channel becomes empty after an input action is performed on the input port connected to it.

In the case where an output port is connected to more than one input port, there is a channel connecting the output port to each input port. All these channels are initially empty. An output action at the source port will cause all the channels to become full. A subsequent output action at that port can only proceed when all the channels that are connected to the port become empty. That is, a subsequent output action can proceed only when all the receiver ports have received the packet. This is done through an input action which causes the connected channel to become empty.

In performing a nondeterminate input action specified by a **from_either** construct, an input action is performed on one of the input ports with a full channel, changing the state of the corresponding channel from full to empty.

9. STRUCTURE SPECIFICATIONS

In PADL, a hierarchy of modules can be constructed by connecting modules (used as submodules) to form higher level modules (composite modules). These submodules are connected together by relating their ports, that is, channels are associated with each pair of source-destination ports connected together.

9.1 Submodule Declarations

All submodules used in a structure specification must be declared. The syntax for a submodule declaration is:

Syntax:

<submod decl> : : = **submodule** <submod decl list>
<submod decl list> : : = <submod decl> { ; <submod decl> } [;]
<submod decl> : : = <submod decl id list> : <mod type name> [(<parameter list>)]
<submod decl id list> : : = <submod decl id> { , <submod decl id> }
<submod decl id> : : = <name> [{ <subscript range list> }]
<subscript range list> : : = <subscript range> { , <subscript range> }
<subscript range> : : = <expression> : <expression>
<mod type name> : : = <name>
<parameter list> : : = <expression>

Each submodule is a named instance of a module type (see Section 10.2). A submodule name may have integer subscripts and submodules of these kind are declared by giving the name and the subscript ranges. In the declarations of submodules, the corresponding module types must be provided. The submodule identifiers are separated from the module types by colons. Submodules of the same module type can be declared as a list of submodule identifiers, separated by commas, with the module type name separated from the last member of the list by a colon. If the parameterized module types are used, the appropriate parameter values must be given in the submodule declaration.

Example:

```
submodule  X : ALU
submodule ADD16 : ADDER(16)
submodule XY_ARRAY{0:15, 0:16} : XY_CELL
submodule A0, A1 : ARBITER; SUB_NETWORK{0:1} : ROUTING_NETWORK(N/2)
```

9.2 The Specification of Connections

The body of text specifying connections are composed of either *basic connection specifications* or *control connection specifications.* Basic connection specifications are used for specifying connections by naming the ports to be logically connected together. Control connection specifications are used for specifying the replication of a group of connections over some subscript range or the conditional selection of connections based on some specified condition.

9.2.1 Basic Connection Specifications

Connections between modules can be specified in two ways – by explicitly specifying the ports that are connected together, or by specifying the ports that are connected to a given module. A connection specification of the first type is termed an *explicit connection* while a specification of the second type is termed an *implicit connection* since one of the pair of connected ports is implied. Two or more ports can be connected together but only one of the ports can be the sender of packets, i.e. it must be of port type **outlet** if the port belongs to a submodule, otherwise it must be of type **inlet**. The ports connected together must be of the same packet type.

Explicit Connection

Syntax:

<explicit conn> ::=<conn port id> -> <conn port id> { , <conn port id> }
<conn port id> ::= [<submodule id> .] <port id>
<submodule id> ::= <name> [{ <subscripts> }]
<subscripts> ::= <expression>

If *submodule id* is absent, then the port belongs to the containing module. The port on the left hand side of "-> " is the sender of packets while those on the right hand side are the receivers.

Examples:

```
BUFFER1.OUTPUT ->  ADDER.OPERAND1
ADDER.SUM ->  OUTPORT
FIFO.DO<0> ->  MULTIPLEXOR.DI<0>, MASTER.DIRECTION
ROUTER{0,0}.OP<0> ->  ROUTER{0,1}.IP<0>
```

The first example above specifies a connection between the port OUTPUT of the submodule BUFFER1 with the port OPERAND1 of ADDER. In the second example, the SUM port of ADDER is connected to the port

OUTPORT of the containing structure, i.e. that one that has adder as one of its submodule. A single-source two-destination connection is shown in the third example. Here the ports DO and DI, respectively, of the submodules FIFO and MULTIPLEXOR are both parameterized. The zeroth parameter port DO<0> of FIFO is the source while the zeroth parameter port DI<0> of MULTIPLEXOR and the port DIRECTION of MASTER are the destinations. The fourth example is more complicated as it shows a connection between two parameterized submodules – ROUTER{0, 0} and ROUTER{0, 1} – both of which have parameterized ports – OP<0> and IP<0>, respectively.

Implicit Connection

Syntax:

<implicit conn> : : = <submodule id> (<port list>)
<port list> : : = <conn port id> { , <conn port id> }

The number of ports in *port list* must be the same as the number of ports defined for the module type of the submodule. Each port in *port list* is paired off with the corresponding port defined for the module type in the order of their occurrence in the header of the module type definition.

Example:

```
ADDER(INPORT1, INPORT2, OUTPORT)
```

In the above example, assume that ADDER has two input and one output ports. The first two ports of ADDER are input ports that receive the operand packets for the addition operation. The sum is produced as a packet at its output port. The implicit connection given states that the ports INPORT1 and INPORT2 of the module that has ADDER as a submodule, are connected to the input ports of ADDER. OUTPORT, the output port of the containing module is connected to output port of ADDER. The implicit connection specification is equivalent to the explicit connection:

```
INPORT1 ->  ADDER.OPERAND1;
INPORT2 ->  ADDER.OPERAND2;
ADDER.SUM ->  OUTPORT;
```

The following example shows an implicit connection specification for parameterized modules. The cell at the I-th and J-th position of the array XY_ARRAY are connected to its four neighbors – left, top, right and bottom.

Example:

```
XY_ARRAY{I, J}( XY_ARRAY{I,J-1}, XY_ARRAY{I-1,J},
                XY_ARRAY{I,J+1}, XY_ARRAY{I+1,J} )
```

9.2.2 Control Connection Specification

There are two types of control connection specifications in PADL – the conditional connection and the iterative connection. The first is used for specifying connections that are to be made according to some condition. The other is used for specifying a regular connection of submodules like an "array" of submodules.

Conditional Connection

Example:

```
if flag > 0 then INPORT ->  MODULE.RIGHT
else INPORT ->  MODULE.LEFT
endif
```

In the above example, if flag is positive, then INPORT is connected to the port RIGHT of MODULE. Otherwise it is connected to the port LEFT of MODULE.

Syntax:

<conditional conn> ::= **if** <condition> **then** <conn group>
{ **elseif** <condition> **then** <conn group> }
[**else** <conn group>]
endif

<condition> ::= <expression>

<conn group> ::= <conn spec> { ; <conn spec> } [;]

<conn spec> ::= <basic conn spec> | <control conn spec>

<basic conn spec> ::= <explicit conn> | <implicit conn>

<control conn spec> ::= <conditional conn> | <iterative conn>

The condition expression after the **if** and **elseif** must be of arity one and of type **bitstr** of length one. The value of the condition must either be true or false. If the condition evaluates to true, then the specification in the **then** branch is used. When the condition evaluates to false the specification in the first **elseif** with a true

condition or an **else** branch is used, if the **elseif** or **else** branch is provided. The conditional connection becomes a null connection if the condition is false and no **elseif** or **else** branches are provided. Since a connection specification can contain conditional connections, such connections can be arbitrarily nested.

The following example illustrates a cascade connection of 16 submodules – CELL{0} to CELL{15}. The first port of each module is an input port while the second is an output port. Except for the first (i.e. CELL{0}) and last (i.e. CELL{15}) submodule, the output of the submodule is connected to the input of the succeeding submodule while its own input is connected to the output of the preceding submodule. For CELL{0} the input port IP of the containing structure is connected to its own input port while the output port of CELL{15} is connected to the output port OP of the containing structure.

Example:

```
if N==0 then CELL{0}(IP, CELL{1}.INP)
elseif N==15 then CELL{15}(CELL{14}.OUTP, OP)
else CELL{N}(CELL{N-1}.OUTP, CELL{N+1}.INP)
endif
```

Iterative Connection

Syntax:

<iterative conn> ::= **for** <control variable> := <limit1> **to** <limit2>
<conn group>
endfor

where *control variable* is a variable of type **integer**, *limit1* and *limit2* are expressions of arity one that evaluate to integers. The limits are evaluated on entry to the **for** construct.

Example:

```
for I := 1 to 6
 CELL{I}(CELL{I-1}.DO, CELL{I+1}.DI)
endfor
```

The iterative specification given above describes the connection between a chain of 8 submodules such that each submodule, except the first (i.e. CELL{0}) and the last (i.e. CELL{7}), is connected to the module before and after it.

9.3 Connection Body

A connection body is a collection of connection specifications for defining the connections of a module. The reserved word **structure** must immediately precede the connection body while the reserved word **endstruct** must immediately follow it.

Syntax:

<connection body> ::= **structure**
<conn group>
endstruct

A complete structure specification for a chain of cells, connected in the fashion illustrated by the example above, has to define what the connections for the cells at the ends of the chain are. The following example shows such a specification. The input port of the first cell, CELL{0}, is connected to the outside world via port INPUT. Similarly the output port of the last cell, CELL{7}, sends packet out to the outside world via port OUTPUT.

Example:

```
structure
for I := 0 to 7
    if I==0 then CELL{I}(INPUT, CELL{I+1}.DI)
    elseif I=7 then CELL{I}(CELL{I-1}.DO, OUTPUT)
    else CELL{I}(CELL{I-1}.DO, CELL{I+1}.DI)
    endif
endfor
endstruct
```

10. DESCRIPTIONS

A PADL description is a complete specification of a system, containing a collection of definitions – data type, function, behavior module type definitions, structure module type definitions and external module type declarations.

Syntax:

<description> : : = <definition> { <definition> }
<definition> : : = <data type def>
| <external function def>
| <behavior module type def>
| <structure module type def>
| <external module type decl>

Since data type definitions are discussed in Section 4.5, only function definitions and module (both behavior and structure) type definitions are described here.

10.1 Function Definitions

A function definition may occur inside another function definition or a module type definition. Functions defined in this way are termed internal functions. An external function is one which is not defined inside a module type or another function definition. Each PADL function is defined by a function definition, which is a piece of text consisting of :

(1) The reserved word **function.**
(2) The "header" containing the function name, information specifying the arity and types of its arguments, and the types of the returned values.
(3) The type definitions used in the function definition. If this is an external function, the declarations of other external functions used appear here also.
(4) The definitions of the internal functions subsidiary to this one. Function definitions may thus be nested arbitrarily.
(5) The expression giving the values to be returned by the function. This is the "body" of the function definition.
(6) The reserved word **endfun.**

Syntax:

<external function def> ::= **function** <function header>
[<type external def part>]
{ <internal function def> }
<expression>
endfun

<internal function def> ::= **function** <function header>
[<data type def part>]
{ <internal function def> }
<expression>
endfun

<type external def part> ::= <type external def> { ; <type external def> } [;]

<type external def> ::= <data type def> | <external function decl>

<data type def part> ::= <data type def> { ; <data type def> } [;]

<data type def> ::= **type** <data type name> = <data type spec>

<external function decl> ::= **external** <function header>

<function header> ::= <function name> (<decl> { ; <decl> }
returns <data type spec> { , <data type spec> })

<function name> ::= <name>

Example:

```
function sum_of_squares (X, Y : integer returns integer)
   X*X + Y*Y
endfun
```

Function definitions may not contain actions.

Only external functions are accessible to other external function definitions.

Optional type definitions may appear after the header to give names to types. These user-defined names may be used anywhere in the function definition, including its own header. The type definitions (and external declarations) are separated from each other by semicolons.

Example:

```
function complex_multiply (X, Y : complex returns complex)
   type complex = record [ re, im : integer ] ;
   record [ re : X.re * Y.re - X.im * Y.im ;
            im : X.im * Y.re + X.re * Y.im ]
endfun
```

10.1.1 The header and value transmission

The list of formal arguments and their type specifications appear in the header between the left parenthesis and the reserved word **returns**. These declarations are separated from each other by semicolons. Each declaration may contain several value names, which are separated from each other by commas.

The scope of the formal arguments is the body of the function (the expression), less any inner constructs which re-introduce the same value name. Their types are as given in the header declarations, and their values are the values of the arguments given at function invocation. The types of the returned values are given in the list of type specifications, separated by commas, appearing after the reserved word **returns**. This list of types must conform to the body. In every invocation of a function, the number and types of the arguments and returned values must match those of the definition.

The meaning of a function invocation is as follows. If the function F is defined by

function F (a_1 : t_1; . . . ; a_N : t_N **returns** s_1; . . . ; s_K)
BODYEXP
endfun

then, assuming the definition is correct and conforms to its invocation, the invocation

F(ARGEXP)

is equivalent to

let a_1 : t_1 . . . a_N : t_N := ARGEXP **in** BODYEXP **endlet**

10.1.2 The EXTERNAL declaration

All functions used in an external function definition that are not defined within the definition must be declared in an **external** declaration. This declaration consists of the reserved word **external** followed by a copy of the function's header, which is used by the translator for type checking.

Example:

```
function F1 ( X : integer returns integer )
external F2 ( Q : integer returns integer ) ;
external F3 ( Q : integer returns integer ) ;
F2(X)+F3(X)
endfun
```

This body of text defines the external function F1. Since it uses the functions F2 and F3, which are not defined here, they must appear in external declarations. (They must be defined in other external function definitions or accessed in a subroutine library.) The external declarations contain the headers for F2 and F3, just as they might appear in the definitions of those two functions. The formal arguments appearing in the headers ("Q" in the preceding example) have no significance; they are included only for syntactic consistency. The intention is that the headers be copied verbatim from the external function definitions of F2 and F3 into the external function definition of F1.

All external declarations must appear following the header of the outermost function definition, even if the functions being declared are used only by internal functions. The external declarations may precede, follow, or be mixed with the type definitions of the outermost function definition.

10.1.3 Inheritance of data, type definitions, and external declarations

A function has access only to the data presented to it in its invocation. No data values are imported from any enclosing function definition. Type definitions made in one function definition are inherited by all functions subsidiary to it. A redefinition in an internal function of a type name already defined in an outer context is not permitted.

All external declarations made in the outermost function definition are inherited by all internal functions.

10.1.4 Scope of function definitions

The scope of an external function definition consists of the whole PADL description except the external function definition, itself. That is, any external function may be invoked from anywhere except within its own function definition. The scope of an internal function consists solely of the immediately enclosing function definition. Note that this precludes any recursion or mutual recursion.

The scope rules for functions and type definitions are illustrated by the following example:

```
function F ( <header> )
external FF ( <header> ) ;
type T = <data type spec> ;
   function G ( <header> )
   type U = <data type spec> ;
      function M ( <header> )
         function N ( <header> )
          <BODY_N>
         endfun                                  % End of function N
       <BODY_M>
      endfun                                     % End of function M
    <BODY_G>
   endfun                                        % End of function G
   function H ( <header> )
      function P ( <header> )
       <BODY_P>
      endfun                                     % End of function P
    <BODY_H>
   endfun                                        % End of function H
 <BODY_F>
endfun                          % end of function F
```

the body of	may invoke functions
F	FF (external), G, H (internal)
G	FF (external), M (internal)
M	FF (external), N (internal)
N	FF (external)
H	FF (external), P (internal)
P	FF (external)

the body and header of	may use defined types
F	T
G	T, U
M	T, U
N	T, U
H	T
P	T

The external function definitions comprising a description may be translated separately. The manner in which their names are used to access them in libraries and the manner in which they are linked into a complete description is dependent on the implementation. No recursive invocations among external or internal functions are permitted.

10.2 Module Type Definitions

There are two kinds of modules in PADL – behavior and structure modules. Behavior modules are modules that specify the actions on the packets at the input and output ports of the modules. On the other hand, a structure module only specifies the interconnection of its submodules and may not contain any actions. It can be used to describe a hierarchy of modules.

A module type definition specifies the class of modules that have similar behavior or structure. Invocation of a module as a submodule of another module involves giving the local name and module type name of the former. Note that only structure module type definitions can invoke other modules as its submodules.

The types of definitions that can occur inside a behavior module type definition are internal function definitions and data type definitions.

10.2.1 Behavior Module Type Definitions

A behavior module type definition is composed of :

(1) A header consisting of the reserved word **type**, the module type name, an optional list of parameters, the reserved word **module**, and a list of ports for connection to the outside world.

(2) The definitions of all internal functions used. The type definitions as well as the declarations of all external functions used in the module type definition may appear here also. The order in which these appear with respect to each other is not important as long as the definitions appear before they are used.

(3) The declarations and initializations of all state variables used.

(4) A repeatedly executed collection of actions (see Chapter 8) delimited by the reserved words **cycle** and **endcycle.**

(5) The reserved word **endmod** to end the module type definition.

The following is a behavior module type definition. A module of this type sends out packets which are the sum of the packets received at its two input ports.

Example: A behavior module type definition

```
type ADDER =
    module (inlet OPERAND1, OPERAND2 : integer; outlet SUM : integer)
      cycle
          let A : integer = from OPERAND1;
              B : integer = from OPERAND2
          in
              send A+B at SUM
          endlet
      endcycle
    endmod
```

Syntax:

<behavior module type def> ::= type <mod type header>
module (<port decl list>)
[<type external def part>] (see function definition)
{ <internal def> }
[<state var decl part>]
cycle

<compound action> { ; <compound action>}

endcycle

endmod

<mod type header> ::= <mod type name> [(<param decl list>)] = **module** (<port decl list>)

<mod type name> ::= <name>

<param decl list> ::= <decl> { ; <decl> }

<port decl list> ::= <port decl sublist> { ; <port decl sublist> }

<port decl sublist> ::= <port type> <port decl> { ; <port decl> }

<port type> ::= **inlet** | **outlet**

<port decl> ::= <port decl id list> : <data type spec>

<port decl id list> ::= <port decl id> { , <port decl id> }

<port decl id> ::= <name> [< <subscript range list> >]

<internal def> ::= <internal function def>

state variable declaration part

Syntax:

<state var decl part> ::= **var** <state var decl> {; <state var decl> }

<state var decl> ::= <decl> [:= <expression>]

<decl> ::= <name> { , <name> } : <data type spec>

Another example of a behavior module type definition:

```
type LEFT_SHIFTER =
    module (inlet CNT : integer; DIN<0:7> : bitstr; outlet DOUT<0:7> : bitstr)
       cycle
            let N : integer = from CNT;
               DI<0:7> : bitstr = from DIN;
               DO<0:7> : bitstr = shift(DI,N)
            in
               send DO at DOUT
            endlet
       endcycle
    endmod
```

The behavior module defined above accepts integer valued packets at the input port CNT. A bit string of length 8 is received as a packet value at the other input port DIN. A new bit string value, DO, is then generated by shifting the bit string from DIN by an amount which is the value of the packet received at CNT.

The shifted bit string is sent out as a packet at DOUT.

10.2.2 Structure Module Type Definitions

A structure module type definition is composed of :

(1) A header consisting of the reserved word **type**, the module type name, an optional list of parameters, the reserved word **module**, and a list of ports for connection to the outside world.

(2) A declaration of the submodules used.

(3) The type definitions used in the module type definition. The declarations of all external functions used appear here also.

(4) The definitions of the internal functions used.

(5) A connection body, i.e. a body of PADL statements specifying the interconnection of submodules (see Chapter 9).

(6) The reserved word **endmod** to indicate the end of the module type definition.

The above description defines a simple connection of two submodules, LEFT and RIGHT, both of type CELL. Each submodule has IP as its input port and OP as its output port. IP of LEFT is connected to the input port IN of the module type PAIR. The output port of LEFT is connected to the input port of RIGHT which in turn has its output port connected to the output port OUT of PAIR.

Example:

```
type PAIR = module (inlet IN : integer; outlet OUT : integer)
            submodule LEFT, RIGHT : CELL;
            structure
            IN -> LEFT.IP
            LEFT.OP -> RIGHT.IP
            RIGHT.OP -> OUT
            endstruct
            endmod
```

Other examples of module type definitions are given in Appendix I.

Syntax:

<structure module type def> : : = **type** <mod type header>
module (<port decl list>)
[<external module decl list>]
[<submod decl>]
[<type external def part>] (same as in function definition)
{ <internal function def> } (same as in function definition)
<connection body>
endmod

10.2.3 EXTERNAL Module Type Declarations

Module types that are invoked in a structure module type definition must be preceded by external module type declarations. External module type declarations are used for type checking of the module parameters and packet types of the ports as well as for ensuring consistency in the use of the modules in connection specifications. An external module type declaration is made of the header (i.e. module type name, parameter and port declarations) of the module type definition with the keyword **type** replaced by the keyword **external**. Note that external module type declarations are used only with structure module type definitions since these are the only definitions that can invoked other modules.

Syntax:

<external module type decl list> : : = <external module type decl> { ; <external module type decl> }
<external module type decl> : : = **external** <mod type header>

Thus for the module PAIR discussed above, the following external declaration must precede the module definition.

Example:

```
external CELL = module (inlet IP : integer; outlet OP : integer)
```

Appendix I - EXAMPLES

The following are examples of PADL descriptions. The first example (Example 1) illustrates an adder module that computes the sum of the packets arriving at its two inputs and sends the sum out as a packet at its output. A more complicated example (Example 2) illustrating a simple ALU module is given next. The next three examples (Examples 3, 4 and 5) are behavioral descriptions of a 2×2 router at various levels of abstractions. Example 3 describes the router as a device that processes packets represented as a record. In Example 4, the packets are 9-bit entities. Example 5 illustrates the use of state variables. A structure module type definition for a network of N×N routers is given in the last example (Example 6).

I.1. Example 1: An Adder Module

The adder module specified below computes the sum of the packets of its two inputs (one packet from each input) and sends the sum out as a packet.

```
%
% Behavior specification of an adder module
%
type ADDER = module (inlet OPERAND1, OPERAND2 : integer; outlet OUT : integer)
        cycle
            let  A : integer = from OPERAND1,
                 B : integer = from OPERAND2
            in
                 send A + B at OUT
            endlet
        endcycle
endmod
```

I.2. Example 2: A Simple ALU Module

The ALU module specified below is a single input, single output module. An input (or operand) packet is composed of an opcode and the values of the two operands. The operations that can be selected are integer multiplication, division, addition, and subtraction. The result packet also contains an error field and a value field. If the value of the opcode field in the input packet does not correspond to any of the four opcodes provided, the error flag in the result packet is set to 1 and the value field in the result packet is set to 0. Otherwise the error flag is set to 0 and the result value is set to the value obtained from the operation. The

tagcase construct is used to select the operation based on the value of the operation field of the input packet.

```
%
% Behavior specification of a single input, single output ALU.
%
type ALU = module (inlet IN : OPERAND_PKT; outlet OUT : RESULT_PKT)
       % The operand and result packets are declared as follows.
       type OPERAND_PKT = record [OPCODE : oneof [IMULT, IDIV, IADD, ISUB]
                                           where IMULT = @09, IDIV = @0A,
                                                 IADD = @0B, ISUB = @0C;
                                  OPERAND1, OPERAND2 : integer ]
       type RESULT_PKT = record [FLAG : bitstr;
                                 RESULT_VALUE : integer]
       cycle
          let ERROR_FLAG : bitstr , RESULT : integer =
              let ICELL : OPERAND_PKT = from IN;
                 A, B : integer = ICELL.OPERAND1, ICELL.OPERAND2;
              in
              tagcase ICELL.OPCODE
                   tag @09 :    '0, A * B;
                   tag @0A :    '0, A / B;
                   tag @0B :    '0, A + B;
                   tag @0C :    '0, A - B;
                   otherwise :  '1, 0;                    % Set error flag
              endtag
              endlet
          in
          send record [FLAG : ERROR_FLAG;
                       RESULT_VALUE : RESULT] at OUT
          endlet
       endcycle
endmod
```

I.3. Example 3: 2X2 Router Module Sending and Receiving Packets

Assuming that VALUE is a defined data type, then a PACKET is defined to be a record composing of an ADDR (address) and V (value) fields. ADDR is of type bitstr[0:7] and V is of type VALUE. The 2X2 router has two input ports, IN0, IN1 and two output ports, OUT0, OUT1 all of packet type, PACKET. The

router receives one packet at a time from one of its ports and reformats the packet before sending it out at an output port that is selected based on the value of the least significant bit of the ADDR field of the received packet. If that bit is '1 then the packet is sent out at the output port OUT1, otherwise it is sent out at OUT0. The address field of the packet is rotated by one before being sent out. The rotation is necessary for routing packets through a network of such routers.

```
% Define a behavior module type, ROUTER,
% with 2 input and 2 output ports.
type ROUTER =
module (inlet IN0, IN1 : PACKET; outlet OUT0, OUT1 : PACKET)
    type PACKET = record [ ADDR : bitstr[0:7]; V : VALUE ];
      cycle
      tagcase X = from_either IN0, IN1
        tag IN0, IN1 : let T : PACKET = record [ ADDR : rotr(X.ADDR,1);
                                                 V : X.V ]
                       in
                       if X.ADDR then send T at OUT1
                       else send T at OUT0 endif
                       endlet
      endtag
      endcycle
endmod
```

I.4. Example 4: 2×2 Router that Processes 9-Bit Entities as Packets

This example treats 9-bit entities as packets. The router establishes a link between a pair of input and output ports based on the value of a bit of the first packet. Each packet has a special bit to indicate if it is the last packet for the link. The link is destroyed after the last packet is sent through and the next packet coming in will be treated as the first packet of another transaction. A new link will be set up for that transaction.

```
type ROUTER =
    module (inlet IN0, IN1 : bitstr[0:8]; outlet OUT0, OUT1 : bitstr[0:8])
      cycle
      let INDIR, OUTDIR : bitstr, X : bitstr [0:8] =
      tagcase Y = from_either IN0, IN1
            tag IN0 : '0, Y[0], Y;
            tag IN1 : '1, Y[0], Y;
      endtag
      in
          begin
          if OUTDIR then send rotr(X[0:7],1)||X[8] at OUT1
          else send rotr(X[0:7],1)||X[8] at OUT0 endif;
          if ~X[8] then
            repeat
             let X = if INDIR then from IN1 else from IN0 endif,
                 T : bitstr[0:8] = rotr(X[0:7],1)||X[8]
             in
               if OUTDIR then send T at OUT1
               else send T at OUT0 endif
             endlet
            until X[8]
          endif;
          end
      endlet
      endcycle
    endmod
```

In this example, INDIR, OUTDIR both of type **bitstr**, and X (of type **bitstr**[0:8]), are set to the values of one of two arms of a **tagcase** expression which is used with a **from_either** action. This action receives a packet from one of the input ports. INDIR is '0 if the packet is from input IN0 and '1 if it is from IN1. X[8] is the last packet indicator, i.e., when it is equal to '1, the packet is the last packet of the transaction. The first packet is sent out before the last packet indicator is used to control the transmission of the remaining packets of the transaction. Subsequent packets of the transaction will be sent to the output port determined by the value of OUTDIR – if OUTDIR is '1 then OUT1 is selected, and if it is '0, OUT0 is selected. The value of OUTDIR is obtained from the first packet coming in after initialization or the next packet after the last packet of the last transaction. For routing through a network, the first 8 bits of every packet is rotated by one bit before being sent out.

I.6. Example 6: An N×N Routing Network

The following is an example of a structure module type definition for a network of 2×2 routers connecting N input ports to N output ports, where N is some positive, nonzero power of 2. N is used in the PADL description as a parameter indicating the number of input or output ports in the network. The ports carry packets of type PACKET which is assumed to be defined somewhere else in a data type definition. There are $\log_2 N$ columns of routers in the network with each column having N/2 routers. The network is specified recursively as a column of N/2 routers and two subnetworks of N/2 × N/2 routers as shown in Figure 1. A conditional is used to check for N=2, if this is the case, then the network is just a 2×2 router. If N>2, then the appropriate connection between the column of N/2 routers and the two subnetworks are specified. Figure 2 shows the full interconnection of an 8 × 8 routing network.

```
%
% MODULE TYPE DEFINITION OF AN NXN ROUTING NETWORK,
% N IS SOME POSITIVE, NONZERO POWER OF 2
%
external ROUTER = module (inlet IN0, IN1 : PACKET; outlet OUT0, OUT1 : PACKET)
type ROUTING_NETWORK(N : integer) =
    module (inlet IP<0:N-1> : PACKET; outlet OP<0:N-1> : PACKET)
      submodule ROUTER{0:(N/2)-1} : ROUTER;
                SUB_NETWORK{0:1} : ROUTING_NETWORK(N/2)
      structure
          if N==2 then ROUTER{0}(IP<0>, IP<1>, OP<0>, OP<1>)
            else for I := 0 to N/2-1
                ROUTER{I}( IP<2*I>, IP<2*I+1>, SUB_NETWORK{0}.IP<I>,
                           SUB_NETWORK{1}.IP<I> )
                SUB_NETWORK{0}.OP<I>  ->   OP<I>
                SUB_NETWORK{1}.OP<I>  ->   OP<I+(N/2)>
                endfor
          endif
      endstruct
    endmod
```

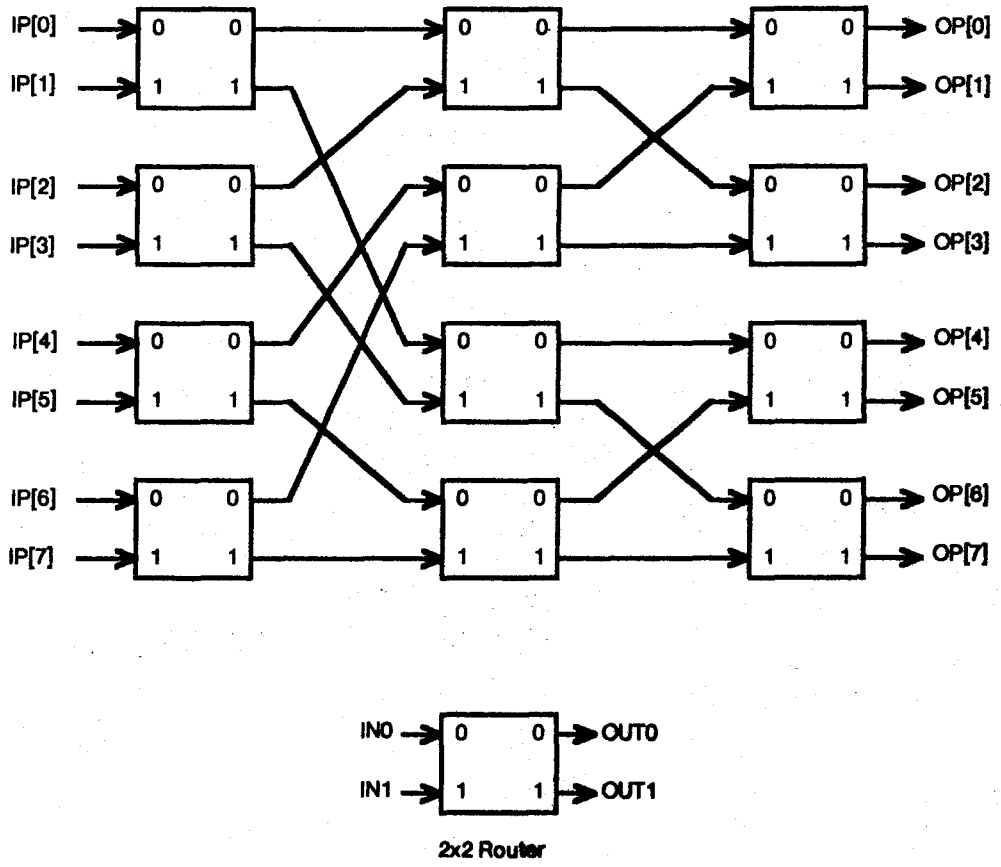

Figure 2. Structure of an 8 by 8 Routing Network

Appendix II - FORMAL SYNTAX

In the following BNF syntax presentation, pairs of large curly braces { ... } indicate zero or more repetitions of the material within, whereas pairs of large square brackets [...] indicate at most one repetition of the material within. All reserved words are printed in **boldface**. A name is a sequence of alphanumeric or underscore characters beginning with a letter.

<description> ::= <definition> { <definition> }
<definition> ::= <module type def>
| <external function def> | <data type def>

<external module type decl> ::= **external** <mod type header>

<module type def> ::= **type** <mod type header>
module (<port decl list>)
<module body>
endmod
<mod type header> ::= <mod type name> [(<param decl list>)] = **module** (<port decl list>)
<mod type name> ::= <name>
<param decl list> ::= <decl> { ; <decl> }
<port decl list> ::= <port decl sublist> { ; <port decl sublist> }
<port decl sublist> ::= <port type> <port decl> { ; <port decl> }
<port type> ::= **inlet** | **outlet**
<port decl> ::= <port decl id list> : <data type spec>
<port decl id list> ::= <port decl id> { , <port decl id> }
<port decl id> ::= <name> [< <subscript range list> >]
<module body> ::= <structure module body> | <behavior module body>
<structure module body> ::= [<external module decl list>]
[**submodule** <submod decl list>]
[<type external def part>]
{ <internal function def> }
<connection body>
<external module type decl list> ::= <external module type decl> { ; <external module type decl> }
<submod decl list> ::= <submod decl> { ; <submod decl> }
<submod decl> ::= <submod decl id list> : <mod type name> [(<parameter list>)]
<submod decl id list> ::= <submod decl id> { , <submod decl id> }
<submod decl id> ::= <name> [{ <subscript range list> }]
<parameter list> ::= <expression>
<behavior module body> ::= [<type external def part>]
{ <internal function def> }

[<state var decl part>]
cycle
<compound action> { ; <compound action> }
endcycle

<external function def> ::= **function** <function header>
[<type external def part>]
{ <internal function def> }
<expression>
endfun

<function header> ::= <function name> (<decl> { ; <decl> } **returns** <data type spec> { , <data type spec> })

<type external def part> ::= <type external def> { ; <type external def> } [;]
<type external def> ::= <data type def> | <external function decl>
<external function decl> ::= **external** <function header>

<internal function def> ::= **function** <function header>
[<data type def part>]
{ <internal function def> }
<expression>
endfun

<data type def part> ::= <data type def> { ; <data type def> } [;]
<data type def> ::= **type** <data type name> = <data type spec>
<data type name> ::= <name>

<data type spec> ::= <basic data type spec> | <compound data type spec> | <data type name>
<basic data type spec> ::= **null** | **integer** | **bitstr** [[<subscript range>]]
<compound data type spec> ::= **array** [<data type spec> <subscript range>]
| **record** [<field spec> { ; <field spec> }]
| **oneof** [<tag spec> { ; <tag spec> }] [**where** <tag def> { , <tag def> }]
<field spec> ::= <field name> { , <field name> } : <data type spec>
<tag spec> ::= <tag name> { , <tag name> } [: <data type spec>]
<tag def> ::= <tag name> { , <tag name> } = <tag value>

<connection body> ::= **structure**
<conn group>
endstruct
<conn group> ::= <conn spec> { ; <conn spec> } [;]
<conn spec> ::= <basic conn spec> | <control conn spec>
<basic conn spec> ::= <explicit conn> | <implicit conn>

<explicit conn> ::=<conn port id> -> <port list>
<implicit conn> ::= <submodule id> (<port list>)
<port list> ::= <conn port id> { , <conn port id> }
<conn port id> ::= [<submodule id> .]<port id>
<submodule id> ::= <name> [{ <subscripts> }]
<control conn spec> ::= <conditional conn> | <iterative conn>
<conditional conn> ::= if <condition> then <conn group>
{ elseif <condition> then <conn group> }
[else <conn group>]
endif
<iterative conn> ::= for <control variable> := <limit1> to <limit2>
<conn group>
endfor
<limit1> ::= <expression>
<limit2> ::= <expression>

<state var decl part> ::= var <state var decl> { ; <state var decl> }
<state var decl> ::= <decl> [:= <expression>]

<compound action> ::= <elementary action>
| <action block>
| <conditional action>
| <tagcase action>
| <iteration>
| <definition block>
<elementary action> ::= <state variable assignment>
| <input action>
| <output action>
<state variable assignment> ::= <state var> { , <state var> } := <actval>
<state var> ::= <name> | <state var array ref> | <state var record ref>
<state var array ref> ::= <state var> [<subscript range> { , <subscript range> }]
<state var record ref> ::= <state var> . <field name>

<input action> ::= from <port id list> | <tagged from>
<tagged from> ::= tagcase [<value name> =] <from-either list> [;]
<tag list> : <expression> [;]
{ <tag list> : <expression> [;] }
[otherwise : <expression> [;]]
endtag
<from-either list> ::= from_either <port id> , <port id list>
<tag list> ::= tag <port id list>

⟨output action⟩ ::= **send** ⟨expression⟩ **at** ⟨port id list⟩
⟨action block⟩ ::= **begin** ⟨compound action⟩ { ; ⟨compound action⟩ } **end**
⟨conditional action⟩ ::= **if** ⟨condition⟩ **then** ⟨compound action⟩ [**else** ⟨compound action⟩] **endif**
⟨tagcase action⟩ ::= **tagcase** [⟨value name⟩ =] ⟨expression⟩ [;]
⟨tag list⟩ : ⟨compound action⟩ [;]
{ ⟨tag list⟩ : ⟨compound action⟩ [;] }
[**otherwise** : ⟨compound action⟩ [;]]
endtag
⟨iteration⟩ ::= **while** ⟨condition⟩ **do** ⟨compound action⟩ | **repeat** ⟨compound action⟩ **until** ⟨condition⟩

⟨definition block⟩ ::= **let** ⟨actdecldef part⟩
in ⟨compound action⟩
endlet
⟨actdecldef part⟩ ::= ⟨actdecldef⟩ { ; ⟨actdecldef⟩ } [;]
⟨actdecldef⟩ ::= ⟨decl⟩
| ⟨def⟩
| ⟨decl⟩ { , ⟨decl⟩ } = ⟨actval⟩
⟨actval⟩ ::= ⟨expression⟩ | ⟨input action⟩

⟨expression⟩ ::= ⟨level1 exp⟩ | ⟨expression⟩ , ⟨level1 exp⟩
⟨level1 exp⟩ ::= ⟨level2 exp⟩ | ⟨level1 exp⟩ | ⟨level2 exp⟩
⟨level2 exp⟩ ::= ⟨level3 exp⟩ | ⟨level2 exp⟩ & ⟨level3 exp⟩
⟨level3 exp⟩ ::= ⟨level4 exp⟩ | ~ ⟨level4 exp⟩
⟨level4 exp⟩ ::= ⟨level5 exp⟩ | ⟨level4 exp⟩ ⟨relational op⟩ ⟨level5 exp⟩
⟨level5 exp⟩ ::= ⟨level6 exp⟩ | ⟨level5 exp⟩ || ⟨level6 exp⟩
⟨level6 exp⟩ ::= ⟨level7 exp⟩ | ⟨level6 exp⟩ ⟨adding op⟩ ⟨level7 exp⟩
⟨level7 exp⟩ ::= ⟨level8 exp⟩ | ⟨level7 exp⟩ ⟨multiplying op⟩ ⟨level8 exp⟩
⟨level8 exp⟩ ::= ⟨primary⟩ | ⟨unary op⟩ ⟨primary⟩

⟨relational op⟩ ::= < | <= | > | >= | == | ~=
⟨adding op⟩ ::= + | -
⟨multiplying op⟩ ::= * | /
⟨unary op⟩ ::= + | -
⟨primary⟩ ::= ⟨constant⟩ | ⟨value name⟩
| (⟨expression⟩)
| ⟨function invocation⟩
| ⟨array ref⟩ | ⟨array generator⟩
| ⟨record ref⟩ | ⟨record generator⟩
| ⟨oneof test⟩ | ⟨oneof generator⟩
| ⟨prefix operation⟩
| ⟨conditional exp⟩

| <letin exp>
| <tagcase exp>
| <forall exp>

<constant> : : = **nil** | **true** | **false** | <integer number> | <bit string constant>

<function invocation> : : = <function name> (<expression>)
<array ref> : : = <primary> [<subscripts>]
<array generator> : : = <primar y> [<subscript range list>]

<record ref> : : = <primary> . <field name>
<record generator> : : = **record** [<field name> : <expression> { ; <field name> : <expression> }]

<oneof test> : : = **is** <tag name> (<expression>)
<oneof generator> : : = **make** <data type spec> [<tag name> : <expression>]

<prefix operation> : : = <prefix operator> (<expression>)
<prefix operator> : : = **abs** | **exp** | **mod** | **shifl** | **shifr** | **rotl** | **rotr** | **bitstr** | **integer**

<conditional exp> : : = **if** <condition> **then** <expression>
{ **elseif** <condition> **then** <expression> }
else <expression>
endif

<letin exp> : : = **let** <decldef part>
in <expression>
endlet

<tagcase exp> : : = **tagcase** [<value name> =] <expression> [;]
<tag list> : <expression> [;]
{ <tag list> : <expression> [;] }
[**otherwise** : <expression> [;]]
endtag

<tag list> : : = **tag** <tag> { , <tag> }
<tag> : : = <tag value> | <tag name>
<tag value> : : = <bit string constant> | <integer number>

<forall exp> : : = **forall** <value name> **in** [<expression>] { , <value name> **in** [<expression>] }
[<decldef part>]
<forall body part>
{ <forall body part> }

endall

⟨forall body part⟩ ::= **construct** ⟨expression⟩ | **eval** ⟨forall op⟩ ⟨expression⟩
⟨forall op⟩ ::= **plus** | **times** | **min** | **max** | **or** | **and**

⟨bit string constant⟩ ::= '⟨bit string⟩ | #⟨octal string⟩ | @⟨hexadecimal string⟩
⟨bit string⟩ ::= ⟨binary char⟩ { ⟨binary char⟩ }
⟨octal string⟩ ::= ⟨octal char⟩ { ⟨octal char⟩ }
⟨hexadecimal string⟩ ::= ⟨hex char⟩ { ⟨hex char⟩ }
⟨binary char⟩ ::= ? | ⟨binary digit⟩
⟨octal char⟩ ::= ? | ⟨octal digit⟩
⟨hex char⟩ ::= ? | ⟨hexadecimal digit⟩

⟨condition⟩ ::= ⟨expression⟩
⟨decldef part⟩ ::= ⟨decldef⟩ { ; ⟨decldef⟩ } [;]
⟨decldef⟩ ::= ⟨decl⟩ | ⟨def⟩ | ⟨decl⟩ { , ⟨decl⟩ } = ⟨expression⟩
⟨def⟩ ::= ⟨name⟩ { , ⟨name⟩ } = ⟨expression⟩
⟨decl⟩ ::= ⟨name⟩ { , ⟨name⟩ } : ⟨data type spec⟩
⟨field name⟩ ::= ⟨name⟩
⟨function name⟩ ::= ⟨name⟩
⟨port id list⟩ ::= ⟨port id⟩ { , ⟨port id⟩ }
⟨port id⟩ ::= ⟨name⟩ [< ⟨subscripts⟩ >]
⟨subscript range list⟩ ::= ⟨subscript range⟩ { , ⟨subscript range⟩ }
⟨subscript range⟩ ::= ⟨expression⟩ : ⟨expression⟩
⟨subscripts⟩ ::= ⟨expression⟩
⟨tag name⟩ ::= ⟨name⟩
⟨value name⟩ ::= ⟨name⟩