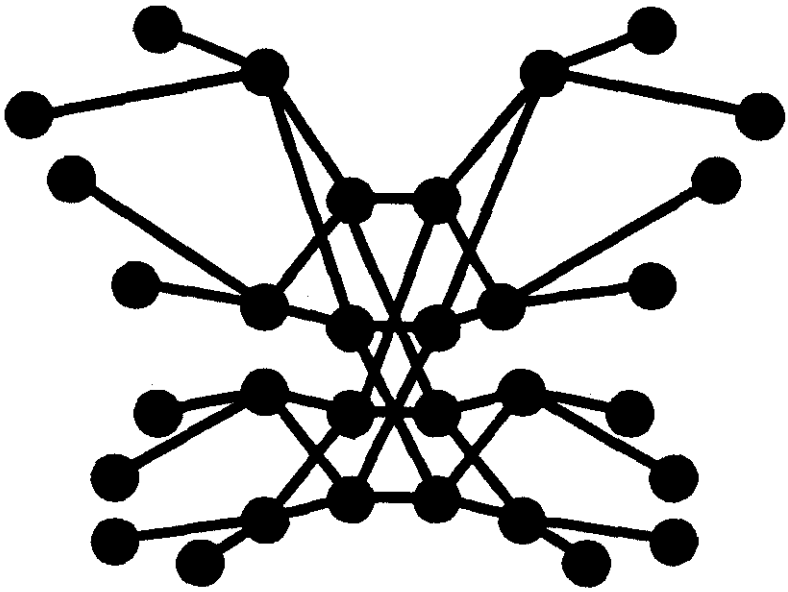

Butterfly Project Report
7

LYNX Reference Manual

Michael L. Scott

Revised Version: August 1986

Computer Science Department
University of Rochester
Rochester, NY 14627

LYNX Reference Manual

Michael L. Scott

March 1986
revised August 1986

ABSTRACT

LYNX is a message-based distributed programming language with novel facilities for communication between processes and for management of context within processes. LYNX was first implemented on the Crystal multicomputer at the University of Wisconsin. It has subsequently been ported to the Butterfly Parallel Processor at the University of Rochester.

This manual is intended for serious users of the Butterfly implementation. At the time of its writing it constitutes the *de facto* standard on the syntax and semantics of LYNX. It also describes ways in which the Butterfly implementation differs from the standard, and in which that implementation resolves issues that the standard leaves undefined.

This work was supported in part by NSF grant number DCR-8320136
and by DARPA contract number DACA76-85-C-0001.

CONTENTS

1. Lexical Conventions

A LYNX program is a sequence of characters from the Ascii character set. Characters are scanned by the compiler from left to right and are grouped together in tokens. Some tokens are valid prefixes of longer tokens. In such cases, the compiler finds tokens of maximal length. Tokens can be separated by white space (spaces, tabs (\t), and newlines (\n)). White space has no other meaning.

Many tokens are simple symbols:

,	;	:	.	..	|	@	
(	)	[	]	{	}	$	
+	-	*	/	%			
<	<=	>=	>	=	<>	×	~>
~	!	&	^	<<	>>		
:=	+:=	-:=	*:=	/:=	%:=		
!:=	&:=	^:=	<<:=	>>:=			

Others are more complicated. All can be defined by regular expressions.

In the following, italics are used for intermediate definitions. Parentheses are used for grouping. Vertical bars are used for alternation. Other adjacent symbols are meant to be concatenated. The function NOT indicates complementation with respect to the Ascii character set. Epsilon (ε) denotes the empty string.

Comments in LYNX begin with '--' and extend through end-of-line.

COMMENT =
  -- (NOT (\n)) *

Comments are treated like white space.

Integer constants can be expressed in octal, decimal, or hexadecimal.

NUMBER =
  0 (0 | *octdigit*) * |
  decdigit (0 | *decdigit*) * |
  # (0 | *hexdigit*) *
where
  octdigit = '1'..'7'
  decdigit = '1'..'9'
  hexdigit = '1'..'9', 'A'..'F', 'a'..'f'

Real number constants can be expressed with or without a scale factor.

REALNUMBER =
  (0 | *decdigit*) + . (0 | *decdigit*) + (*scalefactor* | ε)
where
  scalefactor = (E | e) (+ | - | ε) (0 | *decdigit*) +

Character and string constants are delimited by single and double quotes, respectively. Non-printing characters may be indicated by the single-letter backslash-escapes of C (\b, \n, \r, \t), or by numbers (as defined above) delimited by a *pair* of backslashes (as in \#7f\ for the delete character). Single quotes in character constants and double quotes in string constants are indicated by \' and \", respectively. Backslashes are indicated by \\. Backslashes not accounted for by any of the preceding rules are ignored.[1]

[1] These conventions agree with C except in the form of numeric escapes.

```
CHARCONST =
    '(
        NOT ( ' , \ , \n , nonprint ) |
        \ NOT ( # , 0 , decdigit , \n , nonprint ) |
        \ number \
    )'
STRINGCONST =
    " (
        NOT ( " , \ , \n , nonprint ) |
        \ NOT ( # , 0 , decdigit , \n , nonprint ) |
        \ number \
    ) * "
```

where

nonprint indicates the Ascii characters with codes 0..8, 11..31, and 127
decdigit is as above
number is as defined for the token "number"

Keywords are:

ACCEPT	AND	ANNOUNCE	ARRAY
AWAIT	BEGIN	BIND	CALL
CASE	CONNECT	CONST	DO
ELSE	ELSIF	END	ENTRY
EXCEPTION	EXIT	EXPORT	EXTERNAL
FOREACH	FORWARD	FROM	FUNCTION
HEADER	IF	IMPORT	IN
LIBRARY	LOOP	MODULE	NOT
OF	ON	OR	OTHERWISE
PROCEDURE	PROCESS	RAISE	READ
RECORD	REMOTE	REPEAT	REPLY
RERAISE	RETURN	REVERSE	SET
STARTPROCESS	THEN	TO	TYPE
UNBIND	UNTIL	USE	VAR
WHEN	WHILE	WITH	WRITE

After excluding keywords, identifiers are strings of letters, digits, and underscores that do not begin with a digit and do not end with an underscore. Case is not significant in identifiers, except when significance is imposed from outside by associating names in the language with external objects.

```
IDENTIFIER =
    letter | (
        ( letter | _ )
        ( letter | _ , digit ) *
        ( letter | digit )
    )
```

where

letter = 'A'..'Z', 'a'..'z'
digit = '0'..'9'

2. Types

A type is a set of values and a mapping from those values to representations in memory. Types are useful for restricting the values that can be used in various contexts. Several types are pre-defined. Others are described with type constructors.

type ::= IDENTIFIER | enum_type | subr_type
::= array_type | record_type | set_type

Pointer types will be added in a future version of LYNX.

2.1. Pre-defined Types

integer
consists of as many distinct values as can be represented in a single word. Its values lie in a contiguous range centered approximately at the origin (−2147483648 through 2147483647 on the Butterfly).

char
consists of the Ascii characters. Char variables occupy one byte on the Butterfly.

Boolean
consists of the truth values. *True* and *false* are pre-defined constants of type Boolean. Boolean variables occupy one byte on the Butterfly, where *true* and *false* are represented by 1 and 0, respectively.

real
consists of a very large number of floating point approximations to real numbers. On the Butterfly, real variables occupy 8 bytes and are compatible with *double* variables in C.

link
consists of references to the ends of communication channels. Link values are created at run time. A given end of a given link is accessible to only one process at a time. Links are discussed in detail in the context of communication statements in section 6.15.

Nolink is a pre-defined constant of type link. The value nolink can be assigned into or compared against the contents of a link variable, but is usable for nothing else. Link variables occupy one byte on the Butterfly.

2.2. Enumerations

The values of an enumeration type have a one-one correspondence with the first few non-negative integers.

enum_type ::= (ident_list)
ident_list ::= IDENTIFIER id_list_tail
id_list_tail ::= , ident_list | ε

The identifiers in the list name the values of the type. Enumeration variables occupy four bytes on the Butterfly.

2.3. Subranges

A type can be declared to be a subrange of any existing **scalar** type. The existing type is called the **parent type** of the subrange. Scalar types are integers, chars, Booleans, enumerations, and subranges.

subr_type ::= range
range ::= [expr .. expr]

The range expressions must have values that can be determined at compile time. They cannot involve function calls. Subrange variables occupy one, two, or four bytes on the Butterfly, depending on whether or not their bounds fall in the ranges −128..127, −32768..32767, or −2147483648..2147483647, respectively.

2.4. Array Types

The values of an array type are ordered lists of values of the array's **element type**. The length of each list is the number of distinct values of the array's **index type**.

```
array_type      ::= ARRAY type OF type
```

The type that follows the word ARRAY is the index type. The second type is the element type. The index type must be scalar.

A variable of an array type thus consists of many smaller variables, called the **elements** of the array. The element variables have names: if *expr* is an expression whose type is the index type of array "foo" and whose value is *n*, then "foo [*expr*]" is a name for the *n*th element of foo.

The elements of an array are stored in consecutive locations in memory. The location of the first element is the same as the location of the array. There is no special syntax for multi-dimensional arrays. The programmer can of course declare arrays of arrays and access their elements as "name [row] [column]."

2.5. Record Types

A record is a list of named **fields**. The values of a record type are lists of values of the types of the fields.

```
record_type       ::= RECORD field_list_opt END
field_list_opt    ::= field field_list_opt | ε
field             ::= ident_list : type ; | ;
                  ::= CASE IDENTIFIER : type OF vnt_list_opt END ;
vnt_list_opt      ::= variant vnt_list_opt | ε
variant           ::= { component_list } field_list_opt
component_list    ::= component comp_list_tail
comp_list_tail    ::= , component comp_list_tail | ε
component         ::= expr component_tail
component_tail    ::= .. expr | ε
```

A variable of a record type thus consists of a collection of smaller variables, one for each field of the record. Each of these smaller variables has a name. The name is created by appending a period and the name of the field to the name of the record variable.

The word CASE introduces a variant portion of a record. All variants have the same location. Only one variant is **valid** at a time.

The identifier following the word CASE is the name of a special field called the **tag** of the variant portion of the record. The tag must have a scalar type. It determines which variant is valid at a particular point in time. The component list of each variant lists the values of the tag for which the variant is valid. The lists must be disjoint. Their component expressions must have values that can be determined at compile time. They cannot involve function calls. On the Butterfly, their values must fit in 16 bits.

On the Butterfly, records have the same representation in memory as C structs and unions.

2.6. Set Types

The values of a set type are unordered sets of values of the set's **component type**.

```
set_type        ::= SET OF type
```

The component type must be scalar or link.

On the Butterfly, every set variable occupies 16 bytes. The component type of a set (if other than *link*), must have no more than 128 elements. For sets of subranges of integers, the subrange bounds must lie between 0 and 127, inclusive.

2.7. Pointer Types

The values of a pointer type are references to variables whose type is the pointer's referenced type.

ptr_type ::= @ type

If "foo" is the name of a variable of type @ *ptype*, then "foo@" is a name for the variable of type *ptype* pointed to by foo. Values for pointer variables are created with the unary prefix operator "@" (see section 5.4.2). If foo has not been assigned a value, or if the variable pointed to by foo was a parameter or local variable of a function, procedure, or entry that has since completed execution, then the effect of referring to foo@ is undefined.

On the Butterfly, pointers occupy 4 bytes and have the same representation as in C.

3. Declarations

Identifiers denote constants, types, variables, exceptions, labels, modules, subroutines, entries, libraries, and processes. Several identifiers are pre-defined; all others must be declared by the programmer. Identifiers exported from a module (sections 3.8 and 4) appear in the export list before they are defined. All other identifiers must be defined before they are used.

Labels are declared by their appearance between dollar signs ($) at the beginning of a loop or a compound statement. Variables that are parameters to a process, subroutine, or entry are declared by their appearance in the corresponding header. Variables that are indices for foreach loops are declared by their appearance after the keyword "foreach." All other identifiers must be declared in a **declaration section.**

```
dec_sec       ::= declaration dec_sec | e
declaration   ::= CONST constant_dec const_dec_tail
              ::= TYPE type_dec type_dec_tail
              ::= VAR variable_dec var_dec_tail
              ::= EXCEPTION ident_list ;
              ::= subroutine_hdr ; body ;
              ::= entry_hdr ; body ;
              ::= module ;
```

Declaration sections appear after the headers of processes, libraries, library header files,[2] modules, subroutines, and entries. Within a declaration section, declarations can appear in any order, an arbitrary number of times.

3.1. Use Clauses

Use clauses support separate compilation. A use clause can appear only in the declaration section immediately following the header of a process, library, or library header file. The use clause instructs the compiler to (recursively) include the declarations found in the header files for the libraries named in the identifier list. Identifiers from a header file are treated as if they had been declared at the location of the use clause.

Since a library header file can itself contain use clauses, the compiler may need to search out an arbitrary graph of files. To ensure termination, and to prevent name conflicts when the same file is reached on separate paths, the compiler must ignore second and subsequent uses of the same library. Use clauses are therefore idempotent; the same library can be "used" an arbitrary number of times in the same declaration section. Further details on separate compilation are provided in section 8.

---

[2] The word "header" has two meanings in this manual. To a LYNX programmer, a header is one of the three varieties of program_unit (the other two are the process and the library). This manual uses the phrase "library header file" for this first meaning. It uses the word "header" by itself to describe the introductory line of a process, library, library header file, module, subroutine, or entry.

3.2. Types

A type may have any number of names. There are five built-in types. They all have default names (section 2.1). Additional names, both for built-in types and for constructed types, are introduced by type declarations. Types must have names to be used in type casts (section 5.1) or in the headers for subroutines or entries.

```
type_dec        ::= IDENTIFIER = type ; | ;
type_dec_tail   ::= type_dec type_dec_tail | ε
```

Type checking in LYNX is based on **structural** equivalence. Separate lexical occurrences of a type constructor describe the same constructed type, if the constituent types are the same.

3.3. Constants

Constant declarations introduce names for string constants or for values of pre-defined real or scalar types.

```
constant_dec    ::= IDENTIFIER = expr ; | ;
const_dec_tail  ::= constant_dec const_dec_tail | ε
```

The expression must have a value that can be determined at compile time. It cannot involve function calls. If the expression is a string constant, then the declared constant has the type "ARRAY [0..n] of char," where n is the number of characters in the string. Byte n is a null (Ascii 0).

3.4. Variables

Variable declarations reserve memory locations, introduce names for those locations, and associate types with the names.

```
variable_dec    ::= ident_list : type ; | ;
var_dec_tail    ::= variable_dec var_dec_tail | ε
```

The name of a variable refers either to the location of the variable or to the value stored at that location (its **contents**), depending on context. The type of a variable restricts the values that can be stored at its location. It is a programming error to refer to the contents of a variable before storing a value at its location.

3.5. Exceptions

There are several pre-defined exceptions (see section 7.2). Additional exceptions are introduced by exception declarations. When execution enters the scope of an exception declaration, each identifier in the declaration's identifier list becomes the (only) name for a new exception. Like variables, exceptions in different activations of the same function, procedure, entry, or module are distinct.

Exceptions are used only in when clauses (section 6.11) and raise and announce statements (section 6.12).

3.6. Subroutines

Subroutines are parameterized sequences of statements.

```
subroutine_hdr    ::= PROCEDURE IDENTIFIER arg_list_opt
                  ::= FUNCTION IDENTIFIER arg_list_opt fun_type_opt
arg_list_opt      ::= ( mode formal more_m_formals ) | ε
more_m_formals    ::= ; mode formal more_m_formals | ε
mode              ::= VAR | CONST | ε
formal            ::= ident_list : IDENTIFIER
fun_type_opt      ::= : IDENTIFIER | ε
body              ::= dec_sec compound_stmt IDENTIFIER
                  ::= FORWARD | EXTERNAL
```

The name of the subroutine follows the keyword PROCEDURE or FUNCTION. The identifier at the end of a non-trivial subroutine body must match the name of the subroutine.

The argument list specifies **formal parameters** for the subroutine, together with the modes and type names of those parameters. Within the compound statement of a subroutine body, parameters may be used like any other variables. VAR parameters are passed by reference. Plain parameters are passed by value. The contents of CONST parameters cannot be modified. CONST parameters are passed by reference on the Butterfly.

Functions yield a value whose type name follows the argument list. Procedures do not yield a value.

A FORWARD subroutine body indicates that the subroutine will be declared again later in the same scope, with a non-trivial body. The second declaration must omit the argument list and function type (This restriction may be lifted in future versions of LYNX).

An EXTERNAL subroutine body indicates that the subroutine is external to LYNX and must be found by the linker. On the Butterfly, case is significant in the names of external subroutines.

3.7. Entries

An entry resembles a procedure.

```
entry_hdr         ::= ENTRY IDENTIFIER in_args_opt out_types_opt
in_args_opt       ::= ( formal formal_tail ) | ε
formal_tail       ::= ; formal formal_tail | ε
out_types_opt     ::= : ident_list | ε
```

The name of the entry follows the keyword ENTRY. As with subroutines, the identifier at the end of a non-trivial body must match the name of the entry.

An entry cannot be declared EXTERNAL, but it can be declared FORWARD or REMOTE.

```
body              ::= REMOTE
```

A REMOTE body indicates that the entry may de declared again (minus *in* arguments and *out* types, but with non-trivial body) in the same scope. Unlike FORWARD, REMOTE does not *require* the later declaration.

The *in* arguments and *out* types of an entry are templates for the request and reply messages of a remote operation. REMOTE entry bodies allow the programmer to declare operations provided by other processes, or by accept statements in the current process (see section 6.15.2). Non-trivial entry bodies contain a sequence of statements in which the *in* arguments of the entry can be used like any other variables. Through the use of the **bind** statement (section 6.9), the programmer can arrange for an entry to be executed automatically in response to incoming requests.

The Butterfly implementation limits the size of messages. As of this writing, the limit is 2000 bytes for each request and 2000 bytes for each reply.

3.8. Modules

Modules are an encapsulation mechanism for structuring programs and for limiting the scope of identifiers.

```
module          ::= MODULE IDENTIFIER ; import_pt export_pt
                    dec_sec cpd_stmt_opt IDENTIFIER
import_pt       ::= IMPORT ident_list ; | ε
export_pt       ::= EXPORT ident_list ; | ε
cpd_stmt_opt    ::= compound_stmt | END
```

The compound statement of a module, if it has one, is called the module's **initialization code**. For consistency with the terms for subroutines and entries, it is occasionally called the module's **body** as well. The purpose of import and export lists is explained in the following section.

4. Scope

Declaration sections appear near the beginning of every **block**. Blocks are subroutines, entries, and modules.

Declarations introduce **meanings** for identifiers. Identifiers can have different meanings at different places in a program. The portion of a program in which a particular meaning holds is called that meaning's **scope**. The scope of a meaning extends from the declaration of its identifier to the end of the block in which that declaration appears, with four exceptions:

(1) If a nested block contains a declaration of the same identifier, or if a with statement or labeled statement introduces a new meaning for the identifier (see sections 6.5.4, 6.6, and 6.8), then the scope of the outer meaning does not include the scope of the inner meaning.

(2) A meaning does not extend into any nested module unless that meaning is pre-defined (see section 9), or unless its identifier is explicitly **imported** into the module. The names of the values of an enumeration type are imported into a module automatically when their type is imported. If a subrange of an enumeration type is imported, then only the names of the values spanned by the subrange are imported.

(3) If a module explicitly **exports** an identifier, then the meaning of that identifier extends from its declaration inside the module to the end of the *enclosing* block (subject to exceptions (1) and (2)).

(4) The scope of a label or a foreach loop index extends from its declaration to the end of its compound statement or loop, except when hidden as in (1) above.

Identifiers can be imported or exported repeatedly in a nested chain of modules. Appearances of identifiers in import and export lists are idempotent; the same identifier can appear an arbitrary number of times.

For the purpose of defining scopes, the formal parameters of a subroutine or entry are considered to be part of the declaration section immediately following their argument list. They are not visible in as large a scope as is the name of their subroutine or entry.

Two record types visible at the same point in a program can have fields with the same name. Otherwise, declarations of the same identifier must have disjoint scopes. In particular, simultaneously visible enumeration types cannot have values with the same name (This restriction may be lifted in future versions of LYNX).

The **environment** of a particular thread of control at run time is a mapping from names to their current meanings. New meanings appear whenever control enters a with statement, a labeled statement, or the body of a module, subroutine, or entry. The affected identifiers are the names of the with record fields, the label and/or index of the labeled statement, or the identifiers declared in the immediately preceding declaration section. For a with statement or labeled statement, the

meanings disappear with the completion of the statement. For a subroutine or entry, the meanings disappear with the completion of the body of the block. For a module, the meanings disappear with the completion of the closest enclosing subroutine or entry. They may not be *visible* outside the module, unless they are exported. For any particular thread, the appearance and disappearance of meanings occurs in LIFO order (The same is not true of a process as a whole, as discussed in section 7).

5. Expressions

An expression **evaluates** to a value at run time. Every expression has a type. Expressions are composed of atoms, parentheses, function calls, explicit sets, and operators.

```
expr        ::= un_op expr | expr bin_op expr | ( expr )
            ::= CONSTANT | set | designator arg_opt
```

5.1. Atoms

An atom is an explicit constant, or the name of a constant or variable.

```
designator  ::= IDENTIFIER changeover
changeover  ::= [ expr ] changeover | : IDENTIFIER changeover
            ::= . IDENTIFIER changeover | @ changeover | ε
arg_opt     ::= ε
```

The name of a constant or variable is an expression whose value is the value of the constant or the contents of the variable and whose type is the type of the constant or variable. Within a name, a period indicates selection of a field of a record. Brackets indicate selection of an element of an array. An @ sign indicates dereference of a pointer. A colon indicates a type cast.

Type casts are allowed only on variables. A variable name followed by a type cast is the name of an imaginary variable whose type is specified by the cast, whose location is the same as that of the original variable, and whose value is determined by interpreting the data at that location. That value is implementation-dependent, and may be garbage.

5.2. Set Expressions

A set expression evaluates to a value of type "SET OF *component_type*," where *component_type* is a subrange whose bounds are the lowest and highest possible values of any of the component expressions or ranges. The set type is **provisional** in the sense that it may be **coerced** to another type if context requires it.

```
set         ::= { comp_list_opt }
comp_list_opt ::= component_list | ε
```

The value of the set contains the value of each component expression and all values in each component range.

5.3. Function Calls

The type of a function call is specified in the declaration of the function. The value is obtained by **invoking** the function at run time.

```
arg_opt     ::= ( expr_list )
expr_list   ::= expr expr_list_tail
expr_list_tail ::= , expr_list | ε
```

The expressions in the argument list are called **actual parameters**. They must agree in order and number with the formal parameters of the function. Their types must be **compatible** with the types of the formals. Type compatibility is discussed under assignment statements (section 6.1). A function call with no parameters looks like an atom.

The values of the actual parameters are used as initial values for the formal parameters of the function. Actual parameters corresponding to VAR or CONST formal parameters must be variables. The contents of actual parameters corresponding to VAR formal parameters may be changed by invoking the function. The contents of actual parameters corresponding to value or CONST formal parameters are not changed.

5.4. Operators

All operators are pre-defined. They are represented by the following tokens:

```
+     -     *     /     %     @
<     <=    >=    >     =     <>
~     !     &     ^     <<    >>
NOT   AND   OR    IN    ><    ~>
```

NOT, @, and the tilde (~) are **unary prefix** operators. They have one **operand** each, the expression to the right. The minus and percent signs (– and %) can also be unary operators, if there is no expression to the immediate left. Otherwise, they are **binary infix** operators. Binary operators have two operands: the expressions to their left and right. The rest of the operators in the above list are binary.

5.4.1. Operator Precedence

In the absence of parentheses, operands and operators are grouped together according to the following levels of precedence:

Loosest grouping

```
                OR
               AND
< <= >= > = <> IN >< ~>
      + - (binary) & ! ^
        * / % (binary)
  NOT ~ - (unary) % (unary) @
```

Tightest grouping

Operators of equal precedence associate from left to right.

5.4.2. Operator Semantics

For the purposes of this section, define the **base** of any type except a subrange to be the type itself. Define the base of a subrange to be the base of the subrange's parent type.

NOT
: is a unary operator whose operand must have base type Boolean. "NOT *expr*" is an expression of type Boolean whose value is the negation of the value of *expr*.

AND and OR
: are binary operators whose operands must have base type Boolean. "*expr1* AND *expr2*" and "*expr1* OR *expr2*" are expressions of type Boolean whose values are the logical *and* and *or*, respectively, of the values of the operands. The Butterfly implementation performs short-circuit evaluation.

(unary) –
: is an operator whose operand must have base type integer or real. "– *expr*" is an expression of type integer or real (as appropriate) whose value is the additive inverse of the value of *expr*.

+, −, and *
are binary operators. If neither of their operands is a pointer, then both operands must be sets, or else of base type integer or real. If *expr1* and *expr2* are of base type integer or real, then "*expr1 + expr2*," "*expr1 − expr2*," and "*expr1 * expr2*" are expressions of type integer or real (as appropriate) whose values are the sum, difference, and product, respectively, of the values of the operands. The Butterfly implementation performs integer arithmetic in two's complement with no checks for overflow.

If *expr1* and *expr2* are pointers, then their types must be the same, and "*expr1 − expr2*" is an expression of type integer whose value is the distance between the variables referenced by the pointers, measured in multiples of the size of the variables. If *expr1* is a pointer and *expr2* has base type integer, then "*expr1 − expr2*," "*expr1 + expr2*," and "*expr2 + expr1*" are expressions of the same type as *expr1* whose values are references to the variables (if any) located *expr2* positions before or after the variable pointed to by *expr1*. Pointer arithmetic is intended primarily for use within arrays. The results of arithmetic on pointers to variables that are not elements of the same array are implementation-dependent.

If *expr1* and *expr2* are sets, then "*expr1 + expr2*," "*expr1 − expr2*," and "*expr1 * expr2*" are expressions whose values are the union, difference, and intersection, respectively, of the values of the operands. If neither operand has a provisional type, then the types must be the same, and the type of the expression will be the same as well. If exactly one operand has a provisional type, then it is coerced to the type of the other operand, if possible. The coercion is not permitted if 1) the two operands have different component base types, or 2) the bounds of the component type of the provisional operand do not lie within the bounds of the component type of the other operand. If both operands have provisional types, then the bases of their component types must be the same, and the expression has a new provisional type. The component type of the expression has the same base as the component types of the operands, and its bounds are the minimum and maximum of the bounds of the components of the operands.

/ is a binary operator whose operands must have base type integer or real. If *expr1* and *expr2* have base type real, then "*expr1 / expr2*" is an expression of type real whose value is the quotient obtained in dividing *expr1* by *expr2*. If *expr1* and *expr2* have base type integer, then "*expr1 / expr2*" is an expression of type integer whose value is obtained by truncating the quotient of *expr1* and *expr2* toward zero.

(binary) %
is an operator whose operands must have base type integer. "*expr1 % expr2*" is an expression of type integer whose value is the remainder obtained in dividing *expr1* by *expr2*. The remainder has the same sign as the dividend (in this case *expr1*), and does not depend on the sign of the divisor (*expr2*). ((*expr1 / expr2*) * *expr2*) + (*expr1 % expr2*) = *expr1*.

~ is a unary operator whose operand must have base type integer. "~ *expr*" is an expression of type integer whose value is the bitwise logical negation of the value of *expr*.

&, !, and ^
are binary operators whose operands must have base type integer. "*expr1 & expr2*," "*expr1 ! expr2*," and "*expr1 ^ expr2*" are expressions of type integer whose values are the bitwise logical *and*, (inclusive) *or*, and exclusive *or*, respectively, of the values of the operands.

<, <=, >=, and >
are binary operators whose operands must be sets or pointers, or else have scalar or real base types. If the operands are sets, then the type rules described under "+, −, and *" apply. "*set1* **op** *set2*" is an expression of type Boolean whose value reflects the relationship between the two sets. In the order of the heading above, the operators determine whether *set1* is a proper subset, subset, superset, or proper superset of *set2*.

If the operands are scalars or reals, then their base types must be the same, and "*expr1* **op** *expr2*" is an expression of type Boolean whose value indicates whether *expr1* is less than, less than or equal to, greater than, or greater than or equal to *expr2*.

If the operands are pointers, then their types must be the same, and "*expr1* **op** *expr2*" is an expression of type Boolean whose value indicates whether the location of *expr1@* is less than, less than or equal to, greater than, or greater than or equal to the location of *expr2@*. As with pointer arithmetic, the results of comparing pointers to variables that are not elements of the same array are implementation-dependent.

= and <>
are binary operators whose operands must be sets or pointers, be of type link, or have scalar or real base types. If the operands are sets, then the type rules described under "+, −, and *" apply. If the operands are scalars or reals, then their base types must be the same. If the operands are pointers, then their types must be the same. In all cases, "*expr1* **op** *expr2*" is an expression of type Boolean whose value indicates whether *expr1* and *expr2* have the same value.

⨯ is a binary operator whose operands must have type link. "*expr1* ⨯ *expr2*" (read "*expr1* is connected to *expr2*") is an expression of type Boolean whose value indicates whether the values of *expr1* and *expr2* are references to opposite ends of the same link.

~> is a binary operator whose left operand must have type link and whose right operand must be the name of an entry. "*expr* ~> entryname" is an expression of type Boolean whose value indicates whether the link end referenced by *expr* is **bound** to entryname. (Bindings are discussed in section 6.9.)

IN is a binary operator whose right operand must be a set whose component base type is the same as the base type of the left operand. "*expr1* IN *expr2*" is an expression of type Boolean whose value indicates whether the value of the left operand is an element of the value of the right operand.

(unary) %
is an operator whose operand must be a set. "% *expr*" is an expression of type integer whose value is the set's **cardinality** — the number of elements in *expr*.

@ is a unary operator whose operand must be the name of a variable. (The variable can, of course, be a field of a record, an element of an array, the result of a type cast or pointer dereference, or any combination of the above.) If "foo" is the name of a variable of type *ptype*, then "@ foo" is an expression of type @ *ptype* whose value is a reference to foo.

6. Statements

Statements accomplish the work of a program. They change the contents of variables, send messages, and produce output data on the basis of internal calculations, incoming messages, and input data.

```
stmt            ::= non_reply_stmt | reply
non_reply_stmt  ::= simple_stmt | label_opt labeled_stmt
simple_stmt     ::= communication | io | bind_stmt | await_stmt
                ::= if_stmt | case_stmt | with_stmt | raise_stmt
                ::= return_stmt | exit_stmt
                ::= designator des_stmt_tail
                ::= start_proc_stmt | ε
labeled_stmt    ::= loop_stmt | compound_stmt
```

6.1. Assignment Statement

An assignment statement changes the contents of a variable.

```
des_stmt_tail   ::= assign_op expr
assign_op       ::= := | +:= | -:= | *:= | /:= | %:=
                ::= !:= | &:= | ^:= | <<:= | >>:=
```

The **left-hand side** of the assignment precedes the assignment operator. It must be the name of a

variable. The type of the expression on the **right-hand side** must be **compatible** with the type of the left-hand side. For simple assignment (:=), the value of the expression on the right-hand side replaces the contents of the variable on the left-hand side. For the rest of the so-called **assignment operators**, the statement "*var* **op**:= *expr*" is equivalent to "*var* := *var* **op** *expr*," except that the location of *var* is not computed twice, so execution is faster and side-effects, if any, happen only once.

Every type is compatible with itself (compatibility is reflexive). A subrange and its parent type are compatible with each other. Two subranges are compatible with each other if their parent types are compatible and if their sets of values intersect (Run-time checks may be necessary to guarantee that assignments produce valid values for the left-hand side). A string constant is compatible with any array whose elements have base type char. A long string may be truncated to fill a small array. A short string may be extended with garbage to fill a large array. A provisional set type is compatible with any type it could be coerced to match (Run-time checks may again be necessary). Types not covered by these rules are not compatible.

6.2. Procedure Call

Like a function call, a procedure call provides a set of **actual parameters** to be used for the initial values of the formal parameters of the subroutine. Unlike a function, a procedure yields no value.

```
des_stmt_tail      ::= arg_opt
```

Actual parameters must agree in order and number with the formal parameters of the procedure. Their types must be compatible with the types of the formals. Actual parameters corresponding to VAR or CONST formal parameters must be variables. The contents of actual parameters corresponding to VAR formal parameters may be changed by calling the procedure. The contents of actual parameters corresponding to value or CONST formal parameters are not changed.

6.3. If Statement

An if statement contains one or more lists of statements, at most one of which is executed. The choice between the lists is based on the values of one or more Boolean expressions.

```
if_stmt            ::= IF expr THEN stmt_list_opt elsif_list_opt else_opt END
elsif_list_opt     ::= ELSIF expr THEN stmt_list_opt elsif_list_opt | ε
else_opt           ::= ELSE stmt_list_opt | ε
stmt_list_opt      ::= stmt ; stmt_list_opt | ε
```

The first statement list is executed if the first Boolean is true, the second if the second Boolean is true, and so forth. The last list, if present, is executed if none of the Booleans are true.

6.4. Case Statement

Like an if statement, a case statement contains multiple lists of statements. It is intended for the commonly-occurring situation in which the choice between lists is based on the value of a single variable.

```
case_stmt          ::= CASE expr OF case_list_opt default_opt END
case_list_opt      ::= { component_list } stmt_list_opt case_list_opt | ε
default_opt        ::= OTHERWISE stmt_list_opt | ε
```

The expression following the word CASE must be a scalar. The beginning of each **arm** of the case statement has the same syntax as a set expression. The component lists must be disjoint. The expressions they contain must have values that can be determined at compile time. They cannot involve function calls.

Exactly one of the statement lists must be executed. If the value of the scalar expression is found in one of the component lists, then the immediately following statement list is executed. If the value is not found, then the statement list following the word OTHERWISE (if present) is

executed instead. If the value is not found and the OTHERWISE clause is missing, then an error has occurred and execution must halt.

6.5. Loop Statements

Loop statements cause repetitive execution of a nested list of statements.

loop_stmt ::= while_loop | foreach_loop | repeat_loop | forever_loop

6.5.1. Forever Loop

Execution can only leave a forever loop by means of an exit statement, a return statement, or an exception.

forever_loop ::= LOOP stmt_list_opt END

6.5.2. While Loop

The header of a while loop contains a Boolean expression.

while_loop ::= WHILE expr DO stmt_list_opt END

The expression is evaluated before every iteration of the loop. If its value is true, the statements inside the loop are executed. If it is false, execution continues with the next statement following the loop. If the value of the Boolean expression is false the first time it is examined, then the loop is skipped in its entirety.

6.5.3. Repeat Loop

The footer of a repeat loop contains a Boolean expression.

repeat_loop ::= REPEAT stmt_list_opt UNTIL expr

The expression is evaluated after every iteration of the loop. If its value is false, the statements inside the loop are executed again. If it is true, execution continues with the next statement following the loop. The statements inside a repeat loop are always executed at least once.

6.5.4. Foreach Loop

The header of a foreach loop introduces a new variable called the **index** of the loop.

foreach_loop ::= FOREACH IDENTIFIER IN generator DO stmt_list_opt END
generator ::= set | range | designator | REVERSE reversible_gen
reversible_gen ::= range | designator

The scope of the index is the statement list inside the loop. The type of the index is determined by the loop's **generator**. A generator can be a range of values, a set expression, a name of a set variable, or a name of a scalar type.

The generator produces a sequence of values for the index. The statements inside the foreach loop are executed once for each value. If the generator is a range of values, then the type of the index will be the base type of the bounds of the range (The bounds must have the same base type). The index takes on the values in the range in ascending or descending order, depending on whether the word REVERSE appears in the loop header. The range may be empty, in which case the loop is skipped in its entirety.

If the generator is a set expression or a variable of a set type, then the type of the index is the base type of the components of the set. The index takes on the values of the set in arbitrary order.

If the generator is the name of a scalar type, then that type is the type of the index. The index takes on the values of the type in ascending or descending order, depending on whether the word REVERSE appears in the loop header.

The value of the index can be examined but not changed by the statements in the loop. It cannot appear on the left-hand side of an assignment, nor can it be passed as a VAR parameter to any procedure or function, nor can it appear among the request parameters of an accept statement or the reply parameters of a connect or call statement.

6.6. Exit Statement

A exit statement can only appear inside a loop or inner compound statement (not the body of a subroutine, module, or entry). An exit statement causes control to jump to the statement immediately following the loop or compound statement.

exit_stmt ::= EXIT ident_opt
ident_opt ::= IDENTIFIER | ϵ

Any loop statement or compound statement can be preceded by a label.

label_opt ::= $ IDENTIFIER $ | ϵ

The scope of the identifier in a label is the statement list inside the immediately following loop or compound statement. The identifier in an exit statement must have been introduced in a label. Control jumps to the statement immediately following the labeled statement. If the identifier in the exit statement is missing, then control jumps to the statement immediately following the closest enclosing loop or compound statement.

6.7. Return Statement

A return statement can only appear inside a subroutine.

return_stmt ::= RETURN expr_opt
expr_opt ::= expr | ϵ

If the subroutine is a function, then the return expression must be present, and must be compatible with the type of the function. The function yields the value of the expression, and control returns to the evaluation of the expression in which the function call appeared. If control reaches the end of the body of a function without encountering a return statement, then an error has occurred and execution must halt.

If the subroutine is a procedure, then the return expression must be missing. Control continues with the statement immediately following the procedure call. There is an implicit return statement at the end of the body of every procedure.

6.8. With Statement

A with statement makes it easier and more efficient to access the fields of a record.

with_stmt ::= WITH designator DO stmt_list_opt END

The designator must be the name of a record variable. Within the statement list of the with statement, the fields of the record can be named directly, without preceding them with the designator and a period. The with statement constitutes a nested scope; any existing meanings for the names of the fields will be hidden.

6.9. Bind and Unbind Statements

The bind statement associates link ends with entries and naming environments. The unbind statement undoes associations.

bind_stmt ::= BIND expr_list TO ident_list
::= UNBIND expr_list FROM ident_list

Each expression in the expression list must either be of type link or else be a set of component base type link. Each identifier in the identifier list must be the name of an entry. Each mentioned

link end is bound (unbound) to (from) each mentioned entry. If any of the link values are not valid, then an error has occurred and execution must halt.

Binding and unbinding are idempotent operations when performed by a single thread of control; a thread does no harm by making the same binding twice, or by attempting to break a non-existent binding. **Conflicting** bindings are a run-time error. If two threads attempt to bind the same link end to different instances of the same entry (same entry lexically, but different environments), or if one or more threads attempt to bind the same link end to different entries with the same name, then an error has occurred and execution must halt.

The purpose of bindings is discussed under execution (section 7) below.

6.10. Await Statement

The await statement is used to suspend execution of the current thread of control until a given condition holds.

await_stmt ::= AWAIT expr

The expression must be of type Boolean. The current thread will not continue until the expression is true. If it is false when first encountered, it must be changed by a different thread.

6.11. Compound Statement

A compound statement is a delimited list of statements with an optional set of exception handlers.

compound_stmt ::= BEGIN stmt_list_opt hand_list_opt END
hand_list_opt ::= when_clause hand_list_opt | ε
when_clause ::= WHEN exc_list DO stmt_list_opt
exc_list ::= ident_list | OTHERWISE

Compound statements comprise the bodies of subroutines, modules, and entries. They may also be nested anywhere a statement can occur.

Each exception handler consists of a series of when clauses and a statement list. The identifiers in the when clauses must be the names of exceptions. They must be distinct. When an exception arises, a when clause referring to that exception may be executed in place of the remainder of the compound statement's main statement list. Exceptions are discussed in more detail in section 7.2.

6.12. Raise Statement

Some exceptions occur spontaneously in the course of communication on links (see section 7.2). Others are caused by execution of the raise, announce, or reraise statements.

raise_stmt ::= RAISE IDENTIFIER | ANNOUNCE IDENTIFIER | RERAISE

The identifier following the word RAISE or ANNOUNCE must be the name of an exception. The raise statement causes its exception to arise in the current thread of control. The announce statement causes its exception to arise in every thread that has an active handler for that exception. The reraise statement can appear only in a handler. Its effect is the same as if the exception that triggered execution of the current when clause had been raised in the *absence* of that clause.

6.13. Input/Output Statements

Input and output statements read and write Ascii data on the standard input and output streams. In the Butterfly implementation, these streams are the normal Chrysalis *stdin* and *stdout*.

io ::= WRITE (expr_list) | READ (expr_list)

The parameters of read and write have the same format as those of the *scanf* and *printf* routines in C. The first argument must be a string constant or an array whose elements have base type char.

The rest of the arguments must be scalars or strings. The second and subsequent arguments to read are automatically passed by reference.

6.14. StartProcess Statement

The startprocess statement is used to create new processes (as opposed to new threads of control within the current process).

start_proc_stmt ::= STARTPROCESS (expr_list)

The first argument must be a string constant or an array whose elements have base type char. The implementation uses the string to find an executable version of a LYNX process. The rest of the arguments are actual parameters for the process. They must agree in number, and be compatible with, the formal parameters in the process header (see section 7).

6.15. Communication Statements

Communication statements use links to exchange messages with remote processes.

communication ::= call_stmt | connect_stmt | accept_stmt

6.15.1. Connect and Call Statements

The connect statement requests a remote operation. The call statement invokes a local operation.

call_stmt ::= CALL IDENTIFIER call_args_opt
connect_stmt ::= CONNECT IDENTIFIER call_args_opt ON expr
call_args_opt ::= (call_args) | ϵ
call_args ::= expr_list_opt | expr_list_opt

The identifier following the word CONNECT or CALL must be the name of an entry. The final expression of a connect statement must have type link.

The thread of control that executes a connect or call statement is called a **client**. The client creates a **request** message from the actual parameters of the expression list, sends the message, and waits for a **reply** message. The reply will contain new values for the actual parameters in the second expression list. The request actual parameters must agree in number and order with the formal parameters of the entry whose name follows the word CONNECT or CALL. Their types must be compatible with those of the formals. The reply actual parameters must be the names of variables. They must agree in number and order, and be compatible, with the reply types of the entry.

6.15.2. Accept Statement

The accept statement allows a thread of control to serve a request from some other process for a remote operation.

accept_stmt ::= ACCEPT IDENTIFIER arg_opt ON expr ; n_r_s_list_opt reply
n_r_s_list_opt ::= non_reply_stmt ; n_r_s_list_opt | ϵ
reply ::= REPLY arg_opt

The identifier following the word ACCEPT must be the name of an entry. The expression following the word ON must have type link.

The thread of control that executes an accept statement is called a **server**. The server waits for a request message from a client on the other end of the referenced link. When such a message arrives, it will contain new values for the actual parameters in the designator list. The parameters in that list must agree in number and order, and be compatible, with the parameters of the entry whose name follows the word ACCEPT.

The server executes the statement list and returns a reply message to the client. The actual parameters following the word REPLY must agree in number and order, and be compatible, with the reply types of the entry whose name follows the word ACCEPT. The actuals are packaged together to form the reply message. They are returned to the client on the link specified after the word ON — the same link on which the request message arrived.

The syntax of the portion of an accept statement beginning with the word REPLY is a valid statement in and of itself; therefore the statements inside the *accept* cannot include a *reply*.

6.15.3. Reply Statement

Accept statements provide for the **explicit** receipt of requests for remote operations. Entries provide for **implicit** receipt. Within the body of an entry, the reply portion of an accept statement can appear by itself. Its actual parameters must agree in number and order, and be compatible, with the return types of the entry in which the reply statement occurs. The reply message is returned to the client on the same link on which the request message arrived.

If control reaches the end of an entry without replying, or if the thread of control executing the entry attempts to reply more than once, then an error has occurred and execution must halt.

6.15.4. Communication Rules

Requests in the same direction on the same link are guaranteed to arrive in order. Replies are also ordered, though not necessarily with respect to requests. Messages sent on different links need not arrive in order, even if they involve the same pair of processes. Similarly, local destruction of a link may be noticed at the far end of the link in arbitrary order with respect to events on other links.

If any of the following rules are broken, then an error has occurred and execution must halt.

(1) For all communication statements, the value of the expression that follows the word ON must reference a valid link.

(2) A link end may not be enclosed in a message (see following section) if it is currently bound to an entry, or if it is being used in a connect, accept, or reply statement, or if has been used to receive a request for which a reply is still owed, or if it has already been enclosed in another message, even if that message has yet to be received.

(3) A link end may not appear in a connect, accept, or bind statement if it has been enclosed in an outgoing message, even if that message has yet to be received.

6.15.5. Enclosures

There are no limitations on the data types that can appear in the argument lists of connect, accept, and reply statements. In particular, references to links and data structures that contain references to links can be transferred from one process to another.

If a link variable that references an end of a valid link is enclosed in a request or reply message, then transmission of the message has the side effect of *moving* the referenced end to the receiving process. The contents of the receiver's link variable are changed to be a valid reference to the moved end of the link.

A link end that is enclosed in a message is made inaccessible to the sending process, unless communication is interrupted by an exception. Link variables that referenced the end become dangling references; their contents are no longer valid.

A process can own both ends of a link. If it sends a message to itself on that link, references to any enclosures still become invalid, just as if they had been sent to another process. Link variables that refer to enclosures in call statements or in replies from called entries do *not* become dangling; they remain valid.

7. Execution

There is no notion of "program" in LYNX outside the scope of a process. Processes are autonomous entities that communicate by means of messages.

process ::= PROCESS IDENTIFIER in_args_opt ; dec_sec
cpd_stmt_opt IDENTIFIER .

The formal parameters of a process must have pre-defined types (this restriction may be lifted in future versions of LYNX). Link parameters provide the means for a process to communicate with the rest of the world.

A process begins execution with a single thread of control. The task of that thread is to execute the process's main compound statement. Before doing so, the thread recursively executes the initialization code of any nested modules. In general, a thread of control executes the initialization code of a module immediately before executing the body of the subroutine, module, or entry in which that module is declared.

New threads of control are created by instantiating entries. Entries are instantiated by call statements and by the arrival of messages on link ends bound to entries.

The threads in a process do not execute in parallel. A process continues with a given thread until it **blocks** (Blocking statements are listed in section 7.1). It then switches context to another thread. If no other thread is runnable, the process waits for an **event**. An event is the completion of an outstanding connect or reply statement, or the arrival of an "appropriate" request. If no events are expected, then **deadlock** of threads has occurred and execution must halt. *Events occur only when all threads are blocked.*

The occurrence of an event always allows some thread to continue execution. Only one event occurs at a time. The nature of the event determines which thread runs next. If a connect or reply statement has completed, then the thread that executed that statement can continue. If a request has arrived that matches the operation name and parameter types of an outstanding accept statement, then the thread that executed that statement can continue. If a request has arrived that matches the name and parameter types of an entry to which the message's link has been bound, then a new thread of control is created to execute the entry with initial formal parameter values taken from the message. These last two cases serve to define an appropriate request.

If a request arrives on a link end for which there are no outstanding accept statements or bindings to entries, then consideration of the request is delayed until the next time all threads are blocked (at which point it may be delayed again). If a request arrives on a link end for which there *are* one or more outstanding accept statements or bindings, then the contents of the message must be examined in order to determine whether the request is appropriate. If outstanding accept statements or bindings exist, but the name of the requested operation matches none of them, then the request is inappropriate. The INVALID_OP exception is raised at the connect statement in the client thread of control at the other end of the link. The local process continues to wait for an event.

Since the client and server involved in a remote operation will in general be in different processes, they will share no declarations. Run-time checking is necessary to ensure that they agree on the number, order, and types of request and reply parameters. For two processes to communicate, their respective entry headers must have the same formal parameter types. As in assignment statements and calls to subroutines, the types of request and reply *actual* parameters need not be the same as the types of the formals, so long as they are *compatible* (see section 6.1).

If a client requests an operation that the process on the other end of the link is willing to serve (the operation name matches that of an outstanding accept statement or binding), but the server would disagree about the number, order, or structure of the parameters of the request *or* reply messages, then the request is inappropriate. The TYPE_CLASH exception is raised at the connect statement in the client. As in the case of the INVALID_OP exception, the server process continues to wait for an event.

As mentioned in section 4, the meanings of identifiers visible to a given thread of control come and go in LIFO order. Likewise, storage for the variables accessible to the thread can be allocated and deallocated in LIFO order. Variables declared in a process or library are created when their process is created. Parameters and variables declared local to a subroutine or entry are created when control enters the body of their block. Variables declared immediately inside a module are created when control enters the body of the closest enclosing subroutine, entry, library, or process. Different instantiations of the same subroutine or entry do *not* share local variables.

Since a process may have many suspended threads of control at a given point in time, the variables of a process as a whole cannot be managed on a stack. The creation of a new thread of control in an entry creates a new branch in a run-time environment *tree*. The environment of a thread created with a call statement is similar to that of a procedure; in addition to (new) local variables, it shares the variables in enclosing blocks with its caller. The environment of a thread created in response to a message on a bound link is the same as it would have been if the entry in question had been called locally at the point the bind statement was executed.

Control is not allowed to return from a subroutine whose local variables are still accessible to other threads of control or to potential threads that might be created in response to incoming messages. Similarly, a thread does not terminate when it reaches the end of the body of its entry; it too waits for nested threads to finish. A process terminates only after all its threads have finished. A thread that is waiting for nested threads does so at the very bottom of the block, *after* the word END. Exception handlers for the block are no longer active.

7.1. Blocking Statements

The absence of asynchronous context switches allows the programmer to assume that data structures remain consistent until the current thread of control blocks. A context switch between the threads of a process can occur

(1) at every connect, call, accept, and reply statement,

(2) at every await statement,

(3) whenever the current thread terminates, and

(4) whenever control reaches the end of a subroutine, entry, or process whose local variables remain accessible to other threads or potential threads.

In the absence of exceptions, a thread that resumes execution after a context switch continues with the statement immediately following the statement that blocked. Functions must not contain blocking statements or calls to subroutines whose execution may lead to a blocking statement.

7.2. Exception Handling

Exceptions interrupt the normal flow of control. Several exceptions are pre-defined. In the process of communication on link end L, the following may arise:

INVALID_OP
: Raised at a connect statement when the requested operation is not among those for which there are *accepts* or bindings in the process on the far end of L.

TYPE_CLASH
: Raised at a connect statement when the process on the far end of L is willing to serve the requested operation, but the two processes disagree on the number, order, or types of the request or reply parameters.

LOCAL_DESTROYED
: Raised at connect, accept, or reply statements when the link end referenced by L is destroyed by a thread of control in the local process.

REMOTE_DESTROYED
: Raised at connect, accept, or reply statements when the other end of the link referenced by L

is destroyed by a thread in the process that owns it.

REMOTE_EXC

Raised at a connect statement when a remote operation was being served but the server thread has felt an exception that prevents it from replying. Also raised at a reply statement when the client thread has felt an exception that prevents it from receiving the reply.

Additional pre-defined exceptions may be provided by particular implementations. On the Butterfly, there is a pre-defined exception for each of the standard Chrysalis **throw** codes (IOFATAL, NOMEM, BADHANDLE, CONSISTENCY, FAILED, CHECK, and NOWAY). A throw in an external C routine propagates back into LYNX as a genuine exception. Non-standard throw codes can be bound to LYNX exceptions by use of a significant comment (see the manual page for further details). A throw whose code has not been associated with a LYNX exception is treated like an exception for which the current thread has no handler. Divide-by-zero, bus error, and other "spontaneous" throws that occur inside LYNX programs cannot be caught.

Exceptions occur only when raised or announced explicitly, or when all threads are blocked. Raised exceptions are felt in a single thread. Announced exceptions may be felt in an arbitrary number of threads at once. When an exception arises in a given thread, the handlers (when clauses) of the closest enclosing compound statement are examined in order to see if one of them matches the exception that arose. If a match is found, then the thread is moved to the beginning of the matching handler and is ready to continue. The handler will be executed *in place of* the portion of the compound statement that had yet to be executed when the exception occurred. A WHEN OTHERWISE clause matches any exception.

If the closest enclosing compound statement has no handlers, or if none of them matches the exception, then the exception **propagates** to the handlers of the next enclosing compound statement. If the propagation reaches the compound statement comprising the body of a subroutine, then the exception is raised at the subroutine's point of call, and propagation continues. Any nested threads that still have access to the local variables of the subroutine are aborted (recursively). Likewise any bindings that might create such threads are broken.

The propagation of an exception stops when an appropriate handler is found or when the body of an entry or process is reached. A thread with no appropriate handler is aborted. If propagation escapes the scope of an accept statement, or if an exception remains unhandled in the body of an entry that has not yet replied, then the pre-defined exception REMOTE_EXC is raised at the corresponding connect statement in the process on the other end of the link.

In the absence of exceptions, when all threads are blocked, the occurrence of an event allows exactly one thread to continue. With exceptions, however, more than one thread may be unblocked at once. When a link is destroyed, for example, all threads waiting for the completion of communication on the same end of that link are moved to the beginning of their handlers simultaneously. If one of the threads (the current one, in fact) is executing a function, or a routine that was called (transitively) from a function, than that thread continues first. Otherwise, an arbitrary thread continues first. This rule ensures that thread switches never occur in functions or in anything they call.

The announce statement causes its exception to arise in *all and only* those threads that have an active handler for it. Once felt in a thread, an announced exception propagates like a raised exception. The only difference is that the propagation will always encounter an appropriate handler by the time it reaches the compound statement in which the thread originated. An OTHERWISE handler does not count as active, but one that occurs between a specific handler (which does count) and the point where the thread is blocked will catch the announced exception first.

When a connect, accept, or reply statement is interrupted by a programmer-defined exception, one of the following conditions will hold:

connect

1) The request has not been sent. The process at the other end of the link does not know anything has happened.

2) The request has been received by the process at the far end of the link. It is now being served. The reply message will be discarded when it arrives, and the server will feel the REMOTE_EXC exception.

accept

1) A request has not been received. The process at the other end of the link does not know anything has happened.

2) A request has been received. The connected thread (if it still exists) in the process at the other end of the link will feel the REMOTE_EXC exception.

reply

1) The reply has not been received. The connected thread (if it still exists) in the process at the other end of the link will feel the pre-defined exception REMOTE_EXC. If the server thread attempts to reply again, then an error has occurred and execution must stop.

2) The reply has been received. The process at the other end of the link does not know anything has happened.

In each case 1), enclosed links still belong to the sending process. In each case 2), they now belong to the receiver.

8. Separate Compilation

As described in section 3.1, the use clause allows a program to access a LYNX library. Libraries have syntax similar to that of processes:

library ::= LIBRARY IDENTIFIER ; dec_sec cpd_stmt_opt IDENTIFIER .

Libraries and library header files can themselves contain use clauses. The compound statement at the end of a library is called its **initialization code**. Libraries are initialized in deterministic order.

Before executing its main initialization code, and before initializing nested modules, the initial thread of a process executes the compound statement of each library mentioned in the process's use clauses. Likewise, before executing the initialization code of a library the thread executes the compound statement of each additional use-ed library it has not already attempted to initialize. More precisely, consider a directed graph in which nodes represent compilation units and arcs represent the "uses" relation (A—>B iff A uses B). The initial thread of a process constructs a depth-first search tree, rooted in the process itself, and executes the compound statements of the libraries in preorder.

Each library has an associated header file:

header ::= HEADER IDENTIFIER ; dec_sec END IDENTIFIER .

Header files may not include modules. The bodies of their entries must be missing. The bodies of their subroutines must either be missing or EXTERNAL.

body ::= ε

The declarations found in a header file are included automatically at the beginning of the corresponding library. Subroutines with missing bodies are considered to have been declared FORWARD. Entries are considered to have been declared REMOTE.

The Butterfly compiler expects libraries, headers, and processes to appear in separate files. The name of a library file must be the name of the library, followed by ".x". The name of a header file must be the name of the library, followed by ".h". Type checking is enforced by the linker with naming conventions. The symbol-table name of any variable, subroutine, or entry declared in a header file consists of its LYNX name concatenated with an underscore (_), a series of characters that encode type information, and a second, final underscore. Type clashes between a library and its users result in "undefined symbol" messages from **lnk68**.

To produce an executable program, the object file for a process must be linked with the object files for all the libraries it uses, all the libraries *they* use, and so forth. The Butterfly compiler searches for such files automatically, first in the current directory, then in additional libraries specified on the command line, and finally in so-called "standard" directories.

9. Pre-defined Identifiers

The following identifiers are pre-defined.

types:	Boolean, integer, char, link
constants:	true, false, nolink
exceptions:	TYPE_CLASH, INVALID_OP, REMOTE_EXC, LOCAL_DESTROYED, REMOTE_DESTROYED
functions:	newlink, valid, curlink, idle, exclink
procedures:	destroy

Also, for the Butterfly:

exceptions:	IOFATAL, NOMEM, BADHANDLE, CONSISTENCY, FAILED, CHECK, NOWAY
functions:	exccode, exctext, excval

The types, constants, and exceptions have been discussed elsewhere.

The function "newlink" takes a single reference parameter of type link and yields a value of type link. The parameter and function value return references to the two ends of a new link, created as a side effect.

The function "valid" takes a single value parameter of type link and yields a value of type Boolean. The value indicates whether the parameter accesses an end of a currently valid link that can be used in communication or bindings.

The function "curlink" takes no parameters. It returns a value of type link. The value is a reference to the link on which the request message arrived for the closest lexically-enclosing entry (*not* necessarily the original entry for the current thread of control). If there is no enclosing entry, or if the closest enclosing entry was invoked locally with a call statement, then curlink yields nolink.

The functions "exclink," "exccode," "exctext," and "excval" take no parameters. Within the statement list of a handler for a link-related exception, exclink returns a reference to the offending link. The other three functions provide access to the parameters of a **throw** in an external C routine. Exccode and excval return integers. Exctext returns a pointer to a character (string). Exctext is valid in handlers for ordinary exceptions as well. For these, and for throws with a null throwtext, exctext returns the exception's character-string name, or (in the case of pre-defined exceptions) a default description of the exception's significance.

The function "idle" takes no parameters. It returns a value of type Boolean. The value will be true if there are no events ready to occur on any of the links owned by the current process. The function can be used in a "background" thread to poll for incoming or outgoing messages, for example by inserting

await idle;

at the top of a loop.

The procedure "destroy" takes a single value parameter of type link. It destroys the corresponding link. Variables referencing the link become invalid in the process(es) at both ends. An attempt to destroy a nil or dangling link is a no-op.

A ::= B | C

and

A ::= B
::= C

are shorthand for

A ::= B
A ::= C

Epsilon (ϵ) denotes the empty string.

accept_stmt ::= ACCEPT IDENTIFIER arg_opt ON expr ; n_r_s_list_opt reply
add_op ::= + | - | & | ! | ^ | << | >>
arg_list_opt ::= (mode formal more_m_formals) | ϵ
arg_opt ::= (expr_list) | ϵ
array_type ::= ARRAY type OF type
assign_op ::= := | +:= | -:= | *:= | /:= | %:=
::= !:= | &:= | ^:= | <<:= | >>:=
await_stmt ::= AWAIT expr
bin_op ::= OR | AND | rel_op | add_op | mul_op
bind_stmt ::= BIND expr_list TO ident_list
::= UNBIND expr_list FROM ident_list
body ::= dec_sec compound_stmt IDENTIFIER
::= FORWARD | EXTERNAL | REMOTE | ϵ
call_args ::= expr_list_opt | expr_list_opt
call_args_opt ::= (call_args) | ϵ
call_stmt ::= CALL IDENTIFIER call_args_opt
case_list_opt ::= { component_list } stmt_list_opt case_list_opt | ϵ
case_stmt ::= CASE expr OF case_list_opt default_opt END
changeover ::= [expr] changeover | : IDENTIFIER changeover
::= . IDENTIFIER changeover | @ changeover | ϵ
communication ::= call_stmt | connect_stmt | accept_stmt
comp_list_opt ::= component_list | ϵ
comp_list_tail ::= , component comp_list_tail | ϵ
component ::= expr component_tail
component_list ::= component comp_list_tail
component_tail ::= .. expr | ϵ
compound_stmt ::= BEGIN stmt_list_opt hand_list_opt END
connect_stmt ::= CONNECT IDENTIFIER call_args_opt ON expr
const_dec_tail ::= constant_dec const_dec_tail | ϵ
constant_dec ::= IDENTIFIER = expr ; | ;
cpd_stmt_opt ::= compound_stmt | END
dec_sec ::= declaration dec_sec | ϵ
declaration ::= CONST constant_dec const_dec_tail
::= TYPE type_dec type_dec_tail
::= VAR variable_dec var_dec_tail
::= EXCEPTION ident_list ;
::= subroutine_hdr ; body ;
::= entry_hdr ; body ;
::= module ;
default_opt ::= OTHERWISE stmt_list_opt | ϵ
des_stmt_tail ::= assign_op expr | arg_opt
designator ::= IDENTIFIER changeover

```
else_opt          ::= ELSE stmt_list_opt | ε
elsif_list_opt    ::= ELSIF expr THEN stmt_list_opt elsif_list_opt | ε
entry_hdr         ::= ENTRY IDENTIFIER in_args_opt out_types_opt
enum_type         ::= ( ident_list )
exc_list          ::= ident_list | OTHERWISE
exit_stmt         ::= EXIT ident_opt
export_pt         ::= EXPORT ident_list ; | ε
expr              ::= un_op expr | expr bin_op expr | ( expr )
                  ::= CONSTANT | set | designator arg_opt
expr_list         ::= expr expr_list_tail
expr_list_opt     ::= expr_list | ε
expr_list_tail    ::= , expr_list | ε
expr_opt          ::= expr | ε
field             ::= ident_list : type ; | ;
                  ::= CASE IDENTIFIER : type OF vnt_list_opt END ;
field_list_opt    ::= field field_list_opt | ε
foreach_loop      ::= FOREACH IDENTIFIER IN generator DO stmt_list_opt END
forever_loop      ::= LOOP stmt_list_opt END
formal            ::= ident_list : IDENTIFIER
formal_tail       ::= ; formal formal_tail | ε
fun_type_opt      ::= : IDENTIFIER | ε
generator         ::= set | range | designator | REVERSE reversible_gen
hand_list_opt     ::= when_clause hand_list_opt | ε
header            ::= HEADER IDENTIFIER ; dec_sec END IDENTIFIER .
id_list_tail      ::= , ident_list | ε
ident_list        ::= IDENTIFIER id_list_tail
ident_opt         ::= IDENTIFIER | ε
if_stmt           ::= IF expr THEN stmt_list_opt elsif_list_opt else_opt END
import_pt         ::= IMPORT ident_list ; | ε
in_args_opt       ::= ( formal formal_tail ) | ε
io                ::= WRITE ( expr_list ) | READ ( expr_list )
label_opt         ::= $ IDENTIFIER $ | ε
labeled_stmt      ::= loop_stmt | compound_stmt
library           ::= LIBRARY IDENTIFIER ; dec_sec cpd_stmt_opt IDENTIFIER .
loop_stmt         ::= while_loop | foreach_loop | repeat_loop | forever_loop
mode              ::= VAR | CONST | ε
module            ::= MODULE IDENTIFIER ; import_pt export_pt dec_sec
                      cpd_stmt_opt IDENTIFIER
more_m_formals    ::= ; mode formal more_m_formals | ε
mul_op            ::= % | * | /
n_r_s_list_opt    ::= non_reply_stmt ; n_r_s_list_opt | ε
non_reply_stmt    ::= simple_stmt | label_opt labeled_stmt
out_types_opt     ::= : ident_list | ε
process           ::= PROCESS IDENTIFIER in_args_opt ; dec_sec
                      cpd_stmt_opt IDENTIFIER .
start → program_unit ::= process | library | header
raise_stmt        ::= RAISE IDENTIFIER | ANNOUNCE IDENTIFIER | RERAISE
range             ::= [ expr .. expr ]
record_type       ::= RECORD field_list_opt END
rel_op            ::= <> | < | <= | >= | > | >< | ~> | IN | =
repeat_loop       ::= REPEAT stmt_list_opt UNTIL expr
reply             ::= REPLY arg_opt
return_stmt       ::= RETURN expr_opt
```

```
reversible_gen    ::= range | designator
set               ::= { comp_list_opt }
set_type          ::= SET OF type
simple_stmt       ::= communication | io | bind_stmt | await_stmt
                  ::= if_stmt | case_stmt | with_stmt | raise_stmt
                  ::= return_stmt | exit_stmt
                  ::= designator des_stmt_tail
                  ::= start_proc_stmt | ε
start_proc_stmt   ::= STARTPROCESS ( expr_list )
stmt              ::= non_reply_stmt | reply
stmt_list_opt     ::= stmt ; stmt_list_opt | ε
subr_type         ::= range
subroutine_hdr    ::= PROCEDURE IDENTIFIER arg_list_opt
                  ::= FUNCTION IDENTIFIER arg_list_opt fun_type_opt
type              ::= IDENTIFIER | enum_type | subr_type
                  ::= array_type | record_type | set_type
type_dec          ::= IDENTIFIER = type ; | ;
type_dec_tail     ::= type_dec type_dec_tail | ε
un_op             ::= NOT | - | ~ | % | @
var_dec_tail      ::= variable_dec var_dec_tail | ε
variable_dec      ::= ident_list : type ; | ;
variant           ::= { component_list } field_list_opt
vnt_list_opt      ::= variant vnt_list_opt | ε
when_clause       ::= WHEN exc_list DO stmt_list_opt
while_loop        ::= WHILE expr DO stmt_list_opt END
with_stmt         ::= WITH designator DO stmt_list_opt END
```

NAME

xc – Butterfly LYNX compiler

SYNOPSIS

xc [option or file ...]

DESCRIPTION

Xc is a compiler for the LYNX distributed programming language. *Xc* accepts several types of arguments:

Arguments whose names end with '.x' are taken to be LYNX source files.

Arguments whose names end with '.c68' are taken to be intermediate C source files created by an earlier run of *xc* under the **−C** option.

Arguments whose names end with '.s68' are taken to be intermediate assembler files created by an earlier run of *xc* under the **−S** option.

Arguments whose names end with '.o68' or '.a68' are taken to be object files or object-file archives, respectively.

Header files (for separate compilation of LYNX libraries) have names that end with '.h'. Headers are not specified as arguments, but are parsed automatically at the beginning of the corresponding '.x' library file and whenever they are mentioned in use clauses in other files.

Each '.x', '.c68', or '.s68' file is compiled to produce an object file whose name is the name of the source with '.o68' substituted for the original suffix. All object files are linked together to produce a single executable file, named by default 'a.out.68'. Unless linking is supressed (with the **−C**, **−S**, or **−c** options), the compiler automatically searches for, and links in, the object-file version of each library mentioned in a use clause in one of the '.x' files. Object-file archives specified on the command line are searched (for unresolved references) in the order they appear among the other files. If only one source file is specified, and if no object files or object-file archives are specified, then the '.o68' file produced from the source is deleted automatically (unless deletion is inhibited by the **−h** option).

OPTIONS

The following options are interpreted by *xc*. Options and file arguments can be intermixed freely.

− Compile standard input as if it were a '.x' file appearing at this point in the command line.

−c Stop compilation of each argument after producing '.o68' files.

−C Stop compilation of each argument after producing intermediate '.c68' and '.i' files.

−g Generate symbol table information needed by *dbx*(1). This option is of limited use, since the symbol table will reflect the structure of the intermediate C files, not the LYNX source.

−h Save intermediate files. This option is intended for debugging the compiler. It prevents the automatic deletion of '.c68', '.i', and '.o68' files. It also arranges for an assembler listing to be saved in a '.l68' file. The assembler listing is kept in lieu of a '.s68' file. Unless linking is supressed (with the **−C**, **−S**, or **−c** options), symbol-table information is saved in a pair of files whose names are created by appending the suffixes '.syms' and '.map' to the name of the final executable file, with the original '.68' suffix, if any, deleted. The '.map' file contains a numerical-order name list, created by *nm68*(1). The '.syms' file is created by *splitsyms*(1), for use by *ddt*.

−Idir Add *dir* to (the end of) the list of directories in which to search for library header files, library object files, and include files whose names do not begin with '/'. The current directory is searched before any of the directories on this list. The standard directory /usr/lynx/lib is searched after the directories on this list.

-L Generate program listings on standard output.

-o name Use 'name' instead of 'a.out.68' for the final executable file.

-O Invoke an object-code improver. The **-g** option disables **-O**.

-R Do not generate code to perform run time checking of array subscripts, set elements, and subranges.

-S Stop compilation of each argument after producing intermediate '.s68' files.

-v Provide verbose description of compilation.

-w Suppress warning diagnostics.

The options **-w**, **-L**, and **-R** can be turned on and off by compiler directives in the source code itself. LYNX compiler directives are significant comments. They have the form '--* *os* comment' where the asterisk immediately follows the second hyphen, **o** is the option letter, and **s** is an (immediately following) optional + or − sign. A plus turns the feature on; a minus turns it off. A missing sign implies plus.

Two additional forms of directive are available. The significant comment '--* include *file comment*' is used for source file inclusion. 'File' may not contain white space. The significant comment '--* hint *keyword comment*' is used to make suggestions to the compiler in the spirit of Ada *pragmas*. Both the word 'include' and the word 'hint' can be abbreviated to any prefix. The following keywords are current supported:

sequential
When specified immediately after the header of a forward or recursive procedure, indicates that the routine will not cause context switches and can be optimized to avoid use of the cactus stack.

anyargs Supresses checking of the number and types of initial arguments to the process. Can be specified only if no arguments are declared. Meant for use with the *processargs* standard library.

nothrows
When specified immediately after the header of an external procedure or function, indicates that the routine will not cause any throws that the LYNX program is interested in catching. Results in faster code.

throwcode <number>
Indicates that the most recently-declared LYNX exception should be associated with Chrysalis throw code <number>. Throws of <number> in external routines will be reflected into LYNX as if the exception had been *raised*.

FILES

file.x	source file
file.h	header file
file.c68	intermediate C code file
file.i	intermediate C definitions file
file.s68	intermediate assembler file
file.l68	assembler listing file
file.o68	object file
prog.map	name list output file
prog.syms	symbol table output file for *ddt*.
a.out.68	default executable file
/usr/lynx/installed/xc	compiler
/usr/lynx/lib	standard LYNX library directory

/usr/butterfly/chrys/rel/libcs.a
Butterfly C library archive

See /usr/lynx/installed/paths.c for a list of additional files used by the compiler.

SEE ALSO

cc68(1), lnk68(1), as68(1), dbx(1), prof(1), gprof(1)

M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," TR 183, Department of Computer Science, University of Rochester, January 1986. Revised version to appear in *IEEE Transactions on Software Engineering*, December 1986.

M. L. Scott, "LYNX Reference Manual," BPR 7, Department of Computer Science, University of Rochester, March 1986.

DIAGNOSTICS

Diagnostics produced by *xc* are intended to be self-explanatory. Considerable effort has been devoted to making them more useful than those produced by *cc68*(1). Occasional messages may be produced by the assembler or loader. In particular, type clashes between a LYNX library and its users result in 'undefined symbol' errors from *lnk68*(1).

The symbol-table name of an object defined in a library consists of its LYNX name, concatenated with an underscore (_), a series of characters that encode the object's type, and a second, final underscore. An undefined symbol ending with an underscore indicates either that a routine is missing or else that it is declared differently in different compilation units. If *nm68*(1) reveals a symbol that matches the undefined name up through the second-to-last underscore, then a type clash has probably occurred.

BUGS

Command and source lines longer than about 125 characters are not accepted.

Particularly long sections of straight-line code may cause internal tables to overflow. This limitation can be circumvented by breaking the code into two or more consecutive compound statements.

Syntax errors near the beginning of libraries or near use clauses can confuse the syntax corrector. To eliminate a "Parse Error In Corrected Input" message, fix the syntax error and recompile.

Two libraries that export a variable or exception with the same name will share the same data for those objects.