

# LIL Summary

This document describes LIL a « Lisp-like Implementation Language ».

The phrase **”as in Lisp”** appears throughout this document. In these cases, refer to the documentation on Zetalisp for the exact syntax or semantics of the language feature being discussed.

## LIL Overview

LIL is a language for system programming. It’s main application here is in code for the front end processors. “needs info”. The LIL compiler is written in Lisp.

The best way to understand LIL is as a language with Lisp syntax and Pascal semantics. A LIL source program is a list and the LIL compiler reads the source using the Lisp reader. On the other hand, programs written in LIL operate only on numbers. The concepts of symbols, atoms, and lists are not part of LIL.

LIL is a strongly typed language. It has user-defined types but no user-defined generic operations. Variable references are lexically scoped. Function and type declarations must be at top level; they cannot be nested.

The design and names of various LIL language features were taken directly from Lisp. In particular, LIL programs can contain Lisp macros, whose expansion is handled by the compiler.

## Notation conventions

This section describes how to read the definitions.

- All parentheses that appear are required as shown. Parentheses are not part of the meta-syntactic notation.
- Words in uppercase are keywords in the language. LIL itself has no case requirements.
- Words in lowercase are non-terminals in the definition. Keep looking until you find the expansion. Non-terminals like ”type-identifier” follow the same rules as for ”identifier” and do not have separate expansions in the definitions. When space is limited, the ”id” appears as an abbreviation for ”identifier” and ”expr” for ”expression”.
- Elements in italics are optional and can simply be omitted. When one of the elements is underlined, it **is** the default one.
- Sets of possibilities appear in braces: {}. Within the braces, mutually exclusive possibilities are separated by an upright bar: |. Other possibilities are separated by commas. Some definitions have nested sets of braces.

## LIL concepts and Syntax

### Type declarations

Everything in LIL must have a type. You have to declare *identifiers* as being of a certain type. Every expression has a type. For example, the type of K.ET is the type of its last form. All types have to match or be coercible.

DEFTYPE defines an identifier as the name of a type; DEFGLOBAL declares an identifier as an instance of the type. The compiler permits forward references so you can use a type before defining it. Type declarations do not contain values for initialising objects of the type. Initializing has to be done when you declare the identifier for the object.)

Type definitions must appear at top level. Identifier declarations can appear anywhere and are lexically scoped. LIL has four predefined types: WORD, BYTE, LONG, and BOOLE. LIL has four type constructors for user-defined types. No user-defined generic operations are available.

One class of types in LIL contains the numeric types, WORD, BYT, LONG, and *LIL generic number*. LIL generic number is an internal type that is used for numeric literal until they can be assigned to one of the other numeric types. Generic numbers are coerced to one of the basic numeric types as soon as the appropriate type can be determined.

The type constructors for user-defined types can create two classes of types: aggregate (structure types) and non aggregate (enumeration, pointer, and array types).

**Enumeration** For associating members of a class or category. The type is formed by naming the literals of the type or by providing a subrange from another type.

```
Example: (deftype boole (enumeration false true))
         (deftype byte (enumeration (-128 127)))
         (deftype days (enumeration Sunday Monday Tuesday
Wednesday
Thursday Friday
Saturday))
```

The type definition also declares the literals as global identifiers. The identifiers cannot be redefined.

**Pointer** For providing pointers to user-defined types.

```
(deftype status-ptr-type (pointer status-record-type))
```

**Array** For arrays of any named type or arrays of implicit enumeration, pointer, or array types. The number of dimensions possible for any array is not limited. As is Lisp, all array accesses are zero-based and the number in the dimension spec refers to a number one larger than the index of the last element in the array. The type of the array index must be numeric.

```
(deftype map-type (array world 128))
(deftype counter-array-type (array counter-type (-128
128)))
```

See also the section that collects all of the details about arrays.

**Structure** For collections of objects, each of which can have any type. The objects are called fields.

```
(deftype str-type
  (structure ()
    (next code-symbol)
    (kind symbol-kind)
    (minimum-loc word)
    (maximum-loc word) )
```

See also the section that collects all of the details about structures.

*Formal syntax for types:*

*Type-definition ::= (DEFTYPE type-identifier type-generator)*

*Type-generator ::= { noncomposite-type-gen | composite-type-gen }*

noncomposite-type-gen ::= { WORD  
BYTE  
BOOLE  
LONG  
{ (ENUMERATION literal-id1 . *literal-ids*) |  
(ENUMERATION (min-literal-id max-literal-id)) |  
(POINTER type-id) |  
(ARRAY array-type-gen dim-spec1 . dim-specs)}

array-type-gen ::= { type-id | noncomposite-type-gen }

dim-spec ::= { size | (first last+1) }

composite-type-gen ::= ( STRUCTURE option-list  
(field-id 1 noncomposite-type-gen)  
...  
(field-id n noncomposite-type-gen))

option-list ::= ( (*INCLUDE structure-type-id*), *PRESERVE-ORDER* )

## Declarations

The following constructs declare identifiers:

- **DEFGLOBAL** Declares identifiers for global static variables.
- **DEFMANIFEST** Declares identifiers for global compile time constants.
- **DEFTYPE** Declares type identifiers and literals for enumeration types.
- **DEFUN** Declares formal parameter identifiers for functions.
- **LET** Declares identifiers for lexically scoped local variables.

In **LET**, the identifier being declared must have a type. Either you specify the type explicitly or it inherits its type from the type of the expression used to define it. As in Lisp, **LET** specifies parallel binding of the identifiers; **LET\*** specifies sequential binding.

*Formal syntax for declarations:*

```
declaration ::= { global-declaration | let-declaration |  
                parameter-declaration )
```

```
global-declaration ::= { ( DEFGLOBAL id-declaration . id-declaration) |  
                        (DEFMANIFEST ???) }
```

```
id-declaration ::= ( identifier type-identifier . option-list )
```

```
option-list ::= ( { VOLATILE,  
                  (ADDRESS expression),  
                  (ALIGNMENT {WORD | LONG}),  
                  (INIT expression)  
                  } )
```

```
let-declaration ::= { ( LET binding-form-list body) |  
                    (LET* binding-form-list body) }
```

```
binding-form-list ::= ( binding-form . binding-form )
```

```
binding-form ::= { (identifier expression) |  
                 ((identifier type-id) expression) }
```

## Operators

LIL has a standard set of arithmetic, logical and relational operators.

The arithmetic operators are generic. That is, they work on any numeric operands regardless of their internal type. The types for numeric values are WORD, BYTE, and LONG. Numeric literals are of type LIL-generic-number. (Numeric literals are decimal, not octal. Use #0 as in Lisp to enter literal octal values.) The compiler does coercion on operands to make the types match.

The arithmetic operators:

+  
-  
\*  
//  
\  
MIN  
MAX

Logical operators do bit-wise logical operations on numeric type operands. They return numeric values. The logical operators:

LOGAND  
LOGIOR  
LOGXOR  
LOGNOT  
LSHL  
LSHR  
ASHL  
ASHR  
ROTL  
ROTR

Relational operators take operands of ordered (enumerated) types only and return a value of type boole.

The relational operators:

<  
≤  
=  
≠  
≥  
>

The syntax of all operators is as in Lisp.

## Assignment

The assignment operators are SETQ and PSETQ. You can do whole array or whole structure assignment, so long as the types are compatible. SETQ does assignments sequentially; PSETQ does all of the assignments in parallel. As in Lisp, SETQ returns the value of the last form evaluated. PSETQ returns ????.

For assigning initial values, to identifiers, see LET and the INIT option of DEFGLOBAL.

*Formal syntax for assignment*

```
assignment ::= { (SETQ assignments) |
                (PSETQ assignments) }
```

```
assignments ::= identifier expression . assignments
```

## Expressions

LIL is an expression language. With the notable exception of PROG, each form has a value. Details of expressions that involve array and structure selectors appear in the sections on arrays and structures. The pointer dereferencing operator is @. The value of @identifier is the object that the pointer points to. Its type is the type of the thing pointed to.

```
expression ::= { literal |
                identifier |
                function-application |
                prog-form |
                pointer-dereference |
                (AREF array-identifier dimension-value . dimension-value) |
                (structure-field-id structure-identifier) }
```

## Arrays

An array is simply a collection of objects, all of which have the same type, that are selected according to an element index. You can have arrays of anything, including arrays of arrays and arrays of structures.

*Defining an array type.*

The type of the array index must be numeric. At this time, the sizes of all array dimension specs must be compile-time constants.

```
(deftype map-type (array word 12S))
(deftype conter-array-type (array counter-type (-128 128)))
```

*Declaring an array object.*

```
(defglobal (key-map map-type)
           (alt-map map-type))
```

*Array constructors.*

*Array literals.*

*Array element assignment.*

*Array element assignment looks the same as in Lisp.*

```
(SETF selector-expression expression)

(setf (aref counts bin) 36)
```

*Array element selectors.*

As in Lisp, all array accesses are zero-based and the number in the dimension spec refers to a number one larger than the index of the last element in the array. Array elements are selected by AREF and the appropriate index(es). The type of the index must be numeric. LIL does not provide array slices so you need one index for each dimension of the array.

```
(aref counters 4 27)
(setq t (aref key-map #\meta-m))
```

## Structures

A structure is a heterogeneous collection of objects, called fields. Structure fields can have any type.

*Defining a structure type*

Structures are composed of heterogeneous objects. The compiler usually rearranges structure fields internally so as to make optimal use of storage space for an object of that type. Sometimes you need to ensure that the fields remain in the order in which you declared them (for example, setting up packets for network transmission). PRESERVE-ORDER is an option on a structure type definition for requesting that fields in the object remain in the same order as fields in the definition.

```
(deftype str-type
  (structure ())
  (next code-symbol)
  (kind symbol-kind)
  (minimum-loc word)
  (maximum-loc word) ) )
```

*Declaring a structure object*

*Structure constructors*

*Structure literals*

*Structure field assignment*

```
(setf field-selector expression)
```

*Structure field selectors*

Structure element values are selected by means of selector forms that DEFTYPE creates. The name of the form is the name of the field. ??can't be that simple. need to explain INCLUDE in here too??

## Pointers

Forms for making and testing pointers are provided.

- make-pointer**     *type-name value*     *Special Form*  
Returns a pointer of type *type-name* pointing at *value*, *value* must be coercible to the type pointed to by *type-name*.
- make-null-pointer** *type-name*     *Special Form*  
Returns a pointer of type *type-name* pointing at nothing. All null pointers have the same value, so they may be compared and used as flags.
- null pointer-value**  
Returns true if *pointer-value* is a null pointer, false otherwise.

## Functions

Functions must be declared at top level with DEFUN. For a multiple value return, the function declaration must specify the types for the values being returned.

You can have several different functions with the same name, provided that the formal parameter lists do not have the same types for parameters in corresponding positions. The type of a function is determined by ??does the concept of a function type fit here?? So two functions are the same type if they have formal parameters with the same types.

Unlike Lisp, you can use RETURN to return none, one, or multiple values from a function.

*Formal syntax for functions*

function-declaration ::= (DEFUN name-spec parm-list body)

name-spec ::= { name | (name type-idl . typeids ) }

parm-list ::= (parm-declarations)

parm-declarations :: (identifier type-id . options)

options ::= (MODE { VALUE | REF } )

*Function variables*

??function variables should have a type??

function *function-name*     *Special Form*  
Returns a ??????? to *function-name*.

funcall *function*     &rest *args*  
Calls *function* with *args*. This form will not return a value.

## Prog Forms

As in Lisp, LIL has a prog facility. A prog is a construct that is used for its effect rather than for its value and that is used for altering flow of control.

In spite of the fact that it is designed to be executed for effect, a prog can return a value, either using RETURN or by falling out the bottom (depending on the kind of prog).

For controlling flow of control, some prog forms can include labels and unconditional jumps (gotos) to labels. In addition, RETURN provides a mechanism for leaving the body of prog by some means other than falling out the bottom. This is like a break out of a block in a block-structured language.

LIL provides the following set of prog variants, as in Lisp.

PROG	Does not return a value unless a RETURN that returns a value is encountered. Can include unconditional jumps (GO) to a label. Can include local variable declarations.
GO	Within a PROG, jumps to the specified label.
RETURN	Within a PROG (or DO or LOOP), leaves the PROG body immediately and makes the specified value the value of the PROG.
PROGN	Returns the value of the last form evaluated.
PROG1	Returns the value of the first form evaluated.
PROG2	Returns the value of the second form evaluated.

### *Formal syntax for progs*

```
prog ::= { (PROG
  prog-declaration-list
  prog-body) |
  ( { PROG1 | PROG2 }
  prog-body ) }
```

```
return-form ::= (RETURN expressions)
```

## Conditionals

LIL has the standard Lisp conditional structures, IF, COND, AND, and OR. The semantics are as in Lisp. LIL has SELECT and SELECTQ constructs for conditional execution. SELECTQ allows you to select an alternative based on an expression that is a compile-time constant. SELECT works on expressions that are evaluated at run time.

### *Formal syntax for conditionals*

```
Conditional-expression ::= { if-expression |
  cond-expression |
  and-expression |
  or-expression |
  select-expression }
```

## Iteration

LIL has two main iteration constructs, LOOP and DO.

Functions and Special Forms involving Types

Several forms allow manipulation of LIL types. Their behavior is similar to that of Lisp special forms in that type names must be specified. Since LIL is strongly typed, there is no notion of run-time typing.

**type-size**      *type-name*      *Special Form*  
The number of bytes allocated for objects of type *type-name*.

**array-length** *value*  
The number of elements in *value*. The type of *value* must be some array type.

**structure-offset** *slot-name type-name*      *Special Form*  
The distance in bytes from the beginning of objects of type *type-name* to the beginning of *slot-name*.

**coerce**      *type-name value*      *Special Form*  
*value* treated as type *type-name*.

**default-type**      *value*      *Special Form*  
This form gives a specific type to a generic number. The type chosen is the smallest "machine type" which will hold *value*. If *value* has a type already, the result is simply *value*.

**constant**      *type-name &rest args*      *Special Form*  
*type* must be either an array or structure type. A value of type *type-name* is created with values filled in as specified. For a structure, *args* are pairs of slot names and values. For an array, *args* is simply a list of values. Each value specified must itself be constant.

```
(constant struct-1
  slot-1 123
  slot-2 456
  slot-3 (constant array-1. 0 0 0 0 0 ))
```

```
(constant array-1 1 2 3 4 5)
```