ConLangLang (CLL)

Language Reference Manual

Annalise Mariottini

(aim2120)

Spring 2021

Contents

1 Introduction

ConLangLang (CLL) is a language designed for natural and constructed language synthesis and analysis. Word pattern matching, word set creation, and word categorization are focuses of this language's functionality.

CLL is a functional programming language, its main goal being the the output of language data. Users of this language are encouraged to build structure from existing language data or create new language data from the ground up. The highlights of this language include its regular expressions functionality, user-defined types, dictionary creation, and first-class functions.

2 LRM Syntax

The following formatting styles are used to indicate code in the CLL language:

```
Monolength
Italicized Monolength
```

Terminals (literal characters) are in regular monolength styling and nonterminals (productions) are in italicized monolength styling. Productions are to be expanded into additional nonterminals and terminals through the production rules laid out in the document.

The following format will be used to define these production rules:

$$nonterminal \rightarrow production$$

For a given nonterminal, either additional arrows or the '|' symbol can be used to indicate choice in production. Nonterminals are in *italics* and terminals are unstyled.

There are many nonterminals used throughout this document. Most are contextualized within the section in which they are used. You can find a top-down overview of more productions in 12 Production Rules. In particular, pay attention to the definitions of *type* and *expr*.

3 Lexical Conventions

A CLL program consists of a series of well-defined tokens interpreted from written text. There are 5 types of tokens: identifiers, keywords, literals, operators, and separators. A program is made up of a series of these tokens arranged according to the syntax and semantic conventions of the language. Whitespace and comments are ignored, except where they serve to separate tokens. Whitespace is required to separate tokens that would be otherwise unrecognizable as separate without the whitespace.

3.1 Comments

The following indicate the start and end of ignored comment sections:

Multi-line: *start* → `{#`, *end* → `#}`
Single-line: *start* → `#`, *end* → `\n`

3.2 Separators

In CLL, separators are any characters used alongside other tokens that allow for the syntactical and/or semantic understanding of the statements and expressions of a program. These tokens do not have meaning within themselves, but rather allow for the program to understand the intentions of the language user.

4 Identifiers

Identifiers are used to identify data variables (used to store evaluated data), function variables (used to store functions), type variables (used to define type set), and user-defined types and typedefs.

All variable identifiers must match the following regex pattern:

```
['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

User-defined type identifiers must match the following regex pattern:

```
['A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

User-defined typedef identifiers must match the following regex pattern:

```
'$'['A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

Throughout this document, *id* will refer to a variable identifier, *Id* will refer to a type identifier, and $*Id* will refer to a typedef identifier.

5 Keywords

The following keywords are reserved with predefined meaning and cannot be used for anything other than the compiler's predetermined use.

Types

```
int                    list
float                  dict
bool                   fun
string                 type
regex                  typedef
```

Control Flow

```
match                  default
with                   dowhile
byvalue                if
bytype                 else
```

Values

```
true
false
```

6 Literals

Literals exists for the following types: integer, float, boolean, string, regex, list, dictionary, and function. Literals are evaluated as expressions that contain data (or, in the case of function literals, procedures to output data).

There are 3 literals that contain or input/output other data type: list, dictionary, and function. These function literals will declare these data types using a *type* declaration, which allows for recursive definition of type (e.g. a dict that contains lists of lists of ...). The following are the *type* productions:

$$
\begin{aligned}
\textit{type} &\rightarrow \text{int} \mid \text{float} \mid \text{bool} \mid \text{string} \mid \text{regex} \mid \textit{Id} \\
&\rightarrow \text{list}\langle \textit{type} \rangle \\
&\rightarrow \text{dict}\langle \textit{type}, \textit{type} \rangle \\
&\rightarrow \text{fun}\langle \textit{formallist} : \textit{type} \rangle
\end{aligned}
$$

(We will define *formallist* later.)

6.1 Integer Literals

An integer literal consists of a series of digits `0-9` interpreted as a base-10 number. Any leading zeroes are ignored.

6.2 Float Literals

A float literal is a decimal number `[0-9]+.[0-9]+`. I.e. all decimals must have a decimal point surrounded by integers on both sides.

E.g. `1.0` and `0.1` accepted, but not `1.` or `.1`.

6.3 Boolean Literals

A boolean literal is either the value `true` or `false`.

6.4 String Literals

A string literal is a sequence of ≥ 0 single-byte characters enclosed in single quotes. Since each character has a length of one byte, any language encoding that uses byte-length units works (e.g. ASCII and UTF-8). Note: strings cannot contain the single-quote character `'`.

6.5 Regex Literals

A regex literal is a series of string literals and symbolic conventions enclosed within double quotes. This string is compiled into a regular expression. The symbolic conventions for creating regex literals are listed in Table 1 and follow the POSIX Extended Regular Expression standards. Note: regexes cannot contain the double-quote character `"`.

Example: `"(hello)*!"` matches the language {!, hello!, hellohello!, hellohellohello!, ...}

6.6 List Literals

A list literal is a linked list of identically-typed values and takes the following form:

`<`*type*`>[`*exprlist*`]`

Where *exprlist* is a series of ≥ 0 expressions (of the list's type) separated by commas:

$$\textit{exprlist} \rightarrow \textit{expr}, \textit{exprlist} \mid \textit{expr} \mid \epsilon$$

The following is an example of a list literal: `<int>[1,2,x,3-5]` (assuming `x` in an `int`)

6.7 Dictionary Literals

A dictionary literal is a hashtable mapping of key-value pairs and takes the following form:

`<`*type*`,`*type*`>{`*exprpairlist*`}`

Table 1: Regex Conventions

Symbol	Interpretation
`.` :	Matches any character
`*` :	Matches the preceding expression zero, one or several times (postfix)
`+` :	Matches the preceding expression one or several times (postfix)
`?` :	Matches the preceding expression once or not at all (postfix)
`[..]` :	Character set. Ranges are denoted with `-`, as in `[a-z]`. An initial `^`, as in `[^0-9]`, complements the set. To include a `]` character in a set, make it the first character of the set. To include a `-` character in a set, make it the first or the last character of the set.
`^` :	Matches at beginning of line: either at the beginning of the matched string, or just after a `\n` character.
`$` :	Matches at end of line: either at the end of the matched string, or just before a `\n` character.
| :	Matches the preceding element or the following element.
`(..)` :	Grouping of the enclosed subexpression.
`{n,m}` :	Matches the preceding element at least m and not more than n times.
`{m}` :	Matches the preceding element exactly m times.
`{m,}` :	Matches the preceding element at least m times.
`{,n}` :	Matches the preceding element not more than n times.

Where *exprpairlist* is a series of ≥ 0 expression pairs:

$$exprpairlist \rightarrow exprpair, exprpairlist \mid exprpair \mid \epsilon$$
$$exprpair \rightarrow expr : expr$$

The following is an example dictionary literal: `<int,string>{ 0: 'hello', 1: 'world'}`

6.8 Function Literals

A function literal is a block expression that takes in formal arguments and outputs a given type based upon the expressions within its statement block. A function literal takes the following form:

`<`*formallist* `:`*typeo*`>{` *stmtblock* `}`

Where *formallist* is a series of ≥ 0 *type id* variable declarations and *stmtblock* is a series of ≥ 1 statements separated by semicolons:

$$formallist \rightarrow type\ id\ ,\ formallist \mid type\ id \mid \epsilon$$
$$stmtblock \rightarrow stmt; stmtblock \mid stmt;$$

7 Operators

Operators are expressions used to process either 1 or 2 inputs and produce an output. There are 2 types of operators: type-grouped and untyped. Table 2 indicates operator precedence, type (when applicable), symbol, function, number of inputs, and associativity. When stringing

Table 2: Operators

Prec.	Type	Symbol	Function	Inputs	Assoc.
0		`.`	Child access	2	left
1		(*type*)	Cast	1	right
2	integer, float	`-`	Negation	1	right
		`*`	Multiplication	2	left
		`/`	Division	2	left
		`%`	Remainder	2	left
		`+`	Addition	2	left
		`-`	Subtraction	2	left
	string, list	`^`	Concatenation	2	left
	boolean	`!`	Logical Not	1	right
		`&&`	Logical And	2	left
		`||`	Logical Or	2	left
3	integer, float, bool	`==`	Equals	2	left
		`>`	Greater than	2	left
		`<`	Less than	2	left
		`~=`	Type Equals	2	left
4		`=`	Assignment	2	right

together operations, one can override precedence and associativity defaults by enclosing the desired expression in parentheses:

`(` *`expr`* `)`

7.1 Type-Grouped Operators

Type-grouped operators have known types associated with their operations. There are two formats of type-grouped operators: binary operators and unary operators.

7.1.1 Binary Operators

A binary operator takes in 2 inputs using infix notation:

`expr binop expr`

Both *expr* must be of identical type, even if *binop* is an operator that accommodates multiple types (e.g. addition can only occur between 2 integers or 2 floats, but not an integer and a float).

7.1.2 Unary Operators

A unary operator takes in 1 input in prefix notation:

`unop expr`

7.2 Untyped Operators

There are 3 operators that cannot be grouped by type: child access, cast, and assignment.

7.2.1 Child Access

Child access is an binary operator that takes an expression *expr* that evaluates to a typedef instance variable *id* on the left and a child identifier id_c on the right (id_c must be a declared child of the

typedef variable structure $\$Id$ used to declare id). This operation outputs the value of the child id_c from the id variable on the left.

7.2.2 Cast

Cast is a unary operator that takes an expression *expr* on the right and (if compatible) converts its value to the *type* within the cast symbol. The following are valid types of casts:

1. integer to float
2. float to integer
3. integer, float, bool, or regex to string
4. user-defined type Id to its associated predefined type *type* (or compatible by rules 1, 2, or 3)
5. type *type* to user-defined type Id, when Id is defined as associated with type *type* (or compatible by rules 1, 2, or 3)
6. user-defined type Id_1 to user-defined type Id_2, when both are defined as associated with the same predefined type *type* (or compatible by rules 1, 2, or 3)

A variable can accommodate multiple user-defined types at once (as long as they are of the same associated predefined type). Casting a user-defined type to its associated built-in type causes it to no longer be of the user-defined type.

7.2.3 Type Equals

Type equals is a binary operator that takes an expression on its left and a user-defined type on its right. The return value is *true* if the expression is of the user-defined type and *false* if it is not.

7.2.4 Assignment

Assignment is a binary operator that takes an variable declaration statement on the left and an initialization expression *expr* on the right. It assigns *expr* to the variable declared on the left and outputs the value of *expr*. See 8 Data Variables, 8 Function Assignment, and 10 Type Variables for more information on variable assignment

8 Data Variables

Data variables are used to store literal data in an allocated place in memory to be accessed later. Declaration and initialization of data variables must occur concurrently. Because of this, variable names do not need a type declaration, since they will take on the type of the expression being assigned. Thus, (almost trivially) the format for data variable assignment is as follows:

```
id = expr
```

Data variables are mutable and can be assigned to any value type. The compiler keeps track of variable type changes and will ensure that the variable is only used in instances where the variable's current type is allowed.

Data variable ids are evaluated as expressions. When a primitive data variable is evaluated, its value is copied into the new environment to which it is passed. When a complex data variable is evaluated, a pointer to its address is passed into the new environment.

One special case of a data variable is the typedef instance variable. Typedef instance variables are declared from user-defined typedef variables. Typdef instance variable assignment varies greatly from the initialization pattern above. Typedef instance variables must be preceded by a declaration of their defining typedef $\$Id$ and can only be assigned from a *decllist* of child variable assignments corresponding $\$Id$'s variable structure. However, after declaration and initialization, typedef variables are treated the same as complex data variables (see 10.2 TypeDef).

8.1 Primitive Data Types

Integers, floats, booleans, strings, and regexes are primitive data types, meaning they are the most basic data structures available. A variable that holds a primitive data type is equivalent to a literal of that data type.

8.2 Complex Data Types

Lists and dictionaries are complex data types, meaning that they contain elements of other data types (including complex data types and functions). A complex data variables can be initialized either with a complex data type literal (see 6.6 List Literals and 6.7 Dictionary Literals) or with a modification to or copy of an existing complex data variable. Complex data variables are mutable, but standard library functions encourage users to fold or map complex data structures into new data structure elements.

9 Function Variables

A function variable is used to store a function block into a variable id. The id is simply a pointer to the function literal. It can be called or passed to a map or fold standard library function. Function variables cannot be passed to or returned from user-defined functions.

Functions have ≥ 0 possible data types in the function arguments and exactly 1 output type.

9.1 Function Assignment

The function declaration and initialization is the same as with data variables and is as follows:

```
id = expr
```

The following is an example function the trivially outputs the value that was passed to it:

```
myFun = <int x:int>{ x; }
```

9.2 Function Call

A function call is the utilization of a function variable (whether user-defined or part of the standard library) to evaluate the function block of that variable on a specific input. To use a function call, one must provide input that exactly matches the function definition's formal arguments. The function call evaluates the identified function on that specific input and provides the output.

Function call format is as follows:

```
id(exprlist)
```

Where *exprlist* is a series of ≥ 0 expressions separated by commas:

$$exprlist \rightarrow expr,\ exprlist \mid expr \mid \epsilon$$

The length n of *exprlist* must exactly match the length of $formallist$ of *id*. Ordinally, the type of *expr* at position i of *exprlist* must match the *type* at position i of $formallist$ of *id*.

10 Type Variables

10.1 Type

A type variable holds a set of user-defined types that build upon existing data types. The variable *id* encompasses the entire set, while each type defined is considered a member of that set. Each type member must have an associated built-in data type that defines the type of data to be stored within this new user-defined type. These type members set can be used as any built-in data

type (variable declaration and type matching) and can be initialized in the same way as their associated predefined data type. User types are purely aesthetic and allow the user to categorize variables according to their own designations. User types do not functionally change the value being categorized.

Type variable declaration and type member set initialization must occur simultaneously. The type variable id is uncapitalized, while the type members defined within the type set must be capitalized. Type variables and user types are immutable and cannot be overriden after declaration.

Type declaration and initialization is as follows:

```
type id = { Typelist }
```

Where *Typelist* is a series of ≥ 1 type member declarations:

$$Typelist \rightarrow Id\text{<}type\text{>}, Typelist \mid Id\text{<}type\text{>}$$

Example type declaration and initialization:

```
type foo = { Bar<int> | Baz<bool> | Bap<int> }
```

In the above example, `Bar`, `Baz`, and `Bap` are considered to be of type `foo`.

10.2 TypeDef

A typedef is a user-defined structured object type containing ≥ 0 children variables. Creating a typedef consists of two steps: (1) naming and defining the typedef and (2) declaring and initializing a variable instance of the typedef.

10.2.1 TypeDef Definition

Defining a typedef consists of naming the typedef *Id* and defining its structure by declaring its ≥ 1 children variables. Each child is declared by giving its type and id (each child must have a unique id within the typedef). Upon definition, a typedef is simply a template: it does not hold any actual data (i.e. no memory is taken up to hold the children variables).

Typedef definition is as follows:

```
typedef $Id = { decllist }
```

Where *decllist* is a series of ≥ 1 *type id* variable declarations:

$$decllist \rightarrow type\ id;\ decllist \mid type\ id;$$

10.2.2 TypeDef Variable Assignment

Declaring and initializing a typedef variable consists of using the typedef $\$Id$ in the declaration of a new variable. This instance variable is declared with an id of the form *id* and is initialized by assigning values to all the children declared as the structure of that typedef. (More explicitly: for each child c_i of a given typedef definition, the typedef instance variable must initialize an associated child c_i with an expression that matches the declared type of c_i in the typedef definition.) Once a typedef instance variable is declared and initialized, its type is of $\$Id$ (its defining typedef).

To declare and initialize a typedef variable, the variable declaration and children initialization must occur simultaneously in the following format:

```
$Id id = { childreninit }
```

Where *childreninit* is a series of ≥ 1 $id = expr$ assignments:

$$childreninit \rightarrow id = expr;\ childreninit \mid id = expr;$$

The use of typedef instance variables is identical to that of complex data variables: they are immutable, they are overwritable (old memory discarded once no longer in use), and they are passed by reference.

After a typedef instance variable is created, its children may be accessed with the child access operator along with a specific child's id:

id.id

Example typedef declaration, data variable declaration and initialization, and data access:

```
typedef $MyDef = { string s; float f; };
$MyDef myPi = { s = 'pi'; f = 3.141592 };
myPi.f; # outputs 3.141592
```

11 Control Flow Expressions

A block expression is a special type of expression used to create new scopes and determine control flow of expressions. There are 4 types of block expressions: function, match, if-else, and do-while. Every block contains at least one series of ≥ 1 statements (i.e. a *stmtblock*).

11.1 Variable Scope

A block expression has access to any variables defined in any blocks that it is enclosed within (i.e., it has access to its parent's variables, its parent's parent's variables, and so on). Any variables defined within a given *stmtblock* of a block are limited to use in the scope of that *stmtblock*. If a variable in a block has an identical id to a variable from a higher scope, the value of the lower-scoped variable is used (the higher-scoped variable is "hidden" by the lower-scoped variable) until the end of its scope is reached, at which point the higher-scoped variable becomes visible.

11.2 Expression Value

A block's expression value is defined as the last evaluated expression reached within the block. The type of the block's output is defined at the beginning of the block immediately after the block keyword.

11.3 Function Block

A function literal is a block expression that contains one *stmtblock*. A function literal can be utilized as an anonymous function or assigned to a function variable to be executed at a later point. The evaluated output of this function block is determined upon function call. Function literals are described in 6.8 Function Literals. Creating function variables to store function blocks and using function calls to evaluate function blocks are described under 9 Function Variables.

11.4 Match Block

A match block is a value-matching or type-matching control flow expression. A match block takes in an expression and matches it to a sub-block that is evaluated as the expression value of the match block. A match block takes the following format:

`match:`*`type (expr) matchchoice`*

Where *type* is the output type, *expr* is an input expression and *matchchoice* has two options: matching the input *expr* by value or by type.

$$\begin{aligned} \textit{matchchoice} &\rightarrow \text{byvalue } \{\ \textit{valuematchlist}\ \} \\ &\rightarrow \text{bytype } \{\ \textit{typematchlist}\ \} \end{aligned}$$

A *valuematchlist* is a series of ≥ 1 sub-blocks to be matched with by value. A *typematchlist* is a

series of ≥ 1 sub-blocks to be matched with by user-defined type. They take the following form:

$$\begin{aligned} \textit{valuematchlist} &\rightarrow \textit{valuesub-block};\ \textit{valuematchlist} \mid \textit{valuesub-block}; \\ \textit{valuesub-block} &\rightarrow \textit{valuematchexpr}\ \{\ \textit{stmtblock}\ \} \\ \textit{valuematchexpr} &\rightarrow \textit{expr} \mid \text{default} \end{aligned}$$

$$\begin{aligned} \textit{typematchlist} &\rightarrow \textit{typesub-block};\ \textit{typematchlist} \mid \textit{typesub-block}; \\ \textit{typesub-block} &\rightarrow \textit{typematchexpr}\ \{\ \textit{stmtblock}\ \} \\ \textit{typematchexpr} &\rightarrow \textit{Id} \mid \text{default} \end{aligned}$$

In a *valuematchlist*, a match block matches the input value of *expr* with a sub-block in *valuematchlist*. The *valuematchexpr* of each sub-block in *valuematchlist* is either an *expr* that must be of the same type *type* OR the value `default`, which will match with any input *expr*.

In a *typematchlist*, a match block matches the type of the input *expr* with a sub-block in *typematchlist*. The *typematchexpr* of each sub-block in *typematchlist* is either a user-defined type *Id* OR the value `default`, which will match with any input *expr*.

In either value- or type-matching, if multiple sub-blocks are matched with the input *expr*, the sub-block that is listed first will be the sub-block matched. Also in either case, the last block must be a default block.

Once a sub-block is matched, the *stmtblock* of that sub-block is evaluated and output as the expression value of the encompassing match block.

11.5 If-Else Block

An if-else block is a pair of 2 sub-blocks, one of which will be evaluated depending on the result of a boolean expression. Whichever sub-block is evaluated is output as the expression value of the entire if-else block. The format of an if-else block is as follows:

`if:`*`type`* `(`*`expr`*`) { `*`stmtblock`*` } else { `*`stmtblock`*` }`

Where *type* is the output type and *expr* is a boolean expression. If *expr* evaluates to `true`, the first sub-block's *stmtblock* is evaluated. If *expr* evaluates to `false`, the second sub-block's *stmtblock* is evaluated.

11.6 Do-While Block

A do-while block is a single block that is performed once and then repeatedly while a given boolean expression is evaluated as `true`. The last expression reached in the do-while *stmblock* is evaluated as the output expression value of the do-while loop. The format of a while block is as follows:

`while:`*`type`* `(`*`expr`*`) { `*`stmtblock`*` }`

Where *type* is the output type and *expr* is the conditional boolean expression.

12 Program Structure

A CLL program consist of an series of ≥ 1 statements evaluated in-order until end-of-file (EOF) is reached. Statements include expressions, type variable definition, and typdef variable definition.

On the top-level of a CLL program, variable assignment is generally the most useful top-level expression statement, as every expression at the top-level of a CLL program has its value discarded.

12.1 Production Rules

The following production rule set gives an overview of an entire CLL program grammar:

$$
\begin{aligned}
start &\rightarrow stmtblock \text{ EOF} \\
stmtblock &\rightarrow stmt;\ stmtblock \mid stmt; \\
stmt &\rightarrow expr \mid tassign \mid tdassign \\
tassign &\rightarrow \text{type } id = \{Typelist\} \\
&\quad \text{(...skipping Typelist productions...)} \\
tdassign &\rightarrow \text{typedef } id = \{decllist\} \\
&\quad \text{(...skipping decllist productions...)} \\
expr &\rightarrow (expr) \\
&\quad \rightarrow litint \mid litfloat \mid litbool \mid litstring \\
&\quad \rightarrow litregex \mid litlist \mid litdict \mid litfun \mid litnull \\
&\quad \rightarrow binop \mid unop \mid childaccess \mid cast \mid assign \\
&\quad \rightarrow match \mid ifelse \mid while \mid id \\
&\quad \text{(...skipping lit* productions...)} \\
&\quad \text{(...skipping binop, unop productions...)} \\
&\quad \text{(...skipping childaccess, cast productions...)} \\
assign &\rightarrow type\ id = expr \\
type &\rightarrow \text{int} \mid \text{float} \mid \text{bool} \mid \text{string} \mid \text{regex} \\
&\quad \rightarrow \text{list}\langle type\rangle \mid \text{dict}\langle type,type\rangle \\
&\quad \rightarrow \text{fun}\langle formallist{:}type\rangle \mid Id \\
&\quad \text{(...skipping match, ifelse, while productions...)}
\end{aligned}
$$

All skipped productions have been defined previously in this document.

13 Standard Library

The following are all built-in functions that allow for ease of operation upon data.

13.1 String

`sprint<string s : int>`

Prints the string `s`

`sfold<fun f <type x, string s : type>, type x, string s : type>`

Outputs an accumulated variable `x` after applying `f` to `x` and each character of `s`

`ssize<string s : int>`

Outputs the length of `s`

13.2 Regex

`rematch<regex r, string s : bool>`

Outputs true if `r` matches `s`, otherwise false

`resub<regex r, string s, string t, int n : string>`

Outputs `s` with all matches to `r` in group `n` replaced by `t` (note: group 0 is the whole regex, group 1 is the first grouping, and so on).

13.3 List

`ladd<list<`*type*`> l, `*type*` x : list<`*type*`>>`

Outputs `l` with `x` added to the beginning of `l`

`ladd<list<`*type*`> l, `*type*` x, int n : list<`*type*`>>`

Outputs `l` with `x` added to position `n` of `l` (if `n` is $\geq$ the length of `l`, `x` is placed at the end of `l`)

`lfold<fun f<`*type'* `a, `*type*` x : `*type'*`>, `*type'* `a, list<`*type*`> l : `*type'*`>`

Outputs an accumulated variable `a` after applying `f` to `a` and each element `x` of `l`

`lget<list<`*type*`> l : `*type*`>`

Outputs last element of `l` (throws runtime error if empty)

`lmap<fun f <`*type* `x : `*type*`>, list<`*type*`> l : list<`*type*`>>`

Outputs `l` with `f` applied to each element of `l`

`lmem<list<`*type*`> l, `*type*` x : bool>`

Outputs true if `x` is a member of `l`, otherwise false

`lremove<list<`*type*`> l : list<`*type*`>>`

Outputs `l` with the first element removed (throws runtime error if empty)

`lremove<list<`*type*`> l, int n: list<`*type*`>>`

Outputs `l` with element `n` removed (if `n` is $\geq$ than the length of `l`, then the last element of `l` is removed) (throws runtime error if empty)

`lsize<list<`*type*`> l : int>`

Outputs number of elements in `l`

13.4 Dictionary

`dadd<dict<`*type,type'*`> d, `*type*` x, `*type'* `y : dict<`*type,type'*`>>`

Outputs `d` with the key-value `x:y` pair added

`dfold<fun f<`*type''* `a, `*type*` k, `*type'* `v: `*type''*`>, `*type''* `a, dict<`*type*`,`*type'*`> d : `*type''*`>`

Outputs an accumulated variable `a` after applying `f` to `a` and each key-value pair `k` and `v` of `d` (order of elements is not defined)

`dget<dict<`*type,type'*`> d, `*type*` x : `*type'*`>`

Outputs value of `d` key in `x` (throws runtime error if not found)

`dmap<fun f<`*type* `k, `*type'* `v: `*type'*`, dict<`*type*`,`*type'*`> d : dict<`*type*`,`*type'*`> >>`

Outputs `d` with `f` applied to each key-value pair `k` and `v` of `d` (order of elements is not defined)

`dmem<dict<`*type,type'*`> d, `*type*` x : bool>`

Outputs true if `x` is a key of `d`, otherwise false

`dremove<dict<`*type,type'*`> d, `*type*` x : dict<`*type,type'*`>>`

Outputs `d` with `x`'s key-value pair removed (throws runtime error if key not found)

`dsize<dict<`*type,type'*`> d : int>`

Outputs number of key-pair elements in `d`

`dkeys<dict<`*type,type'*`> d : list<`*type*`>>`

Outputs a List of type *type* containing all the keys of `d`

14 Example Program

```
words = <string>['esti', 'lerni', 'havi', 'pano', 'pilko'];
type verb = { Root<string>, Infinitive<string>, Present<string>,
    Past<string>, Future<string> };

typedef $Stems = {
    string inf;
    string pres;
    string past;
    string fut;
};

typedef $VerbDef = {
    Infinitive inf;
    Present pres;
    Past past;
    Future fut;
};

$Stems stems = {
    inf = 'i';
    pres = 'as';
    past = 'is';
    fut = 'os';
};

conj = <string,regex>{ stems.inf: "[a-z]+(i)$", stems.pres: "[a-z]+(as)$",
    stems.past: "[a-z]+(is)$", stems.fut: "[a-z]+(os)$" };

root = <list<Root> l, string w : list<Root>> {
    w = match:string (true) byvalue {
        rematch(dget(conj, stems.inf), w) {
            resub(dget(conj, stems.inf), w, '', 1);
        }
        rematch(dget(conj, stems.pres), w)  {
            resub(dget(conj, stems.pres), w, '', 1);
        }
        rematch(dget(conj, stems.past), w)  {
            resub(dget(conj, stems.past), w, '', 1);
        }
        rematch(dget(conj, stems.fut), w)  {
            resub(dget(conj, stems.fut), w, '', 1);
        }
        default  {
            '';
        }
    };
    if:list<Root> (ssize(w) > 0) {
        ladd(l,(Root)w);
    } else {
        l;
    };
};

create_verb = <dict<Root,$VerbDef> d, Root w: dict<Root,$VerbDef>> {
    $VerbDef v = {
        inf = (Infinitive)((string)w ^ stems.inf);
```

```
        pres = (Present)((string)w ^ stems.pres);
        past = (Past)((string)w ^ stems.past);
        fut = (Future)((string)w ^ stems.fut);
    };
    dadd(d, w, v);
};

words = lfold(root, <Root>[], words);
verb_dict = lfold(create_verb, <Root,$VerbDef>{}, words);

type pronoun = { First<string>, Second<string>, Third<string>, Singular<string>,
    Plural<string>, Fem<string>, Masc<string> };
typedef $PronounDef = {
    list<First> fst;
    Second snd;
    list<Third> thrd;
};
type pronounCase = { Personal<$PronounDef>, Accusative<$PronounDef> };

first_suffix = <list<First> l, string suf : list<First>> {
    addsuf = <First p:First> {
        (First)((string)p ^ suf);
    };
    lmap(addsuf, l);
};

third_suffix = <list<Third> l, string suf : list<Third>> {
    addsuf = <Third p:Third> {
        (Third)((string)p ^ suf);
    };
    lmap(addsuf, l);
};

Personal personal = {
    fst = <First>[(First,Singular) 'mi', (First,Plural) 'ni'];
    snd = (Second,Singular,Plural) 'vi';
    thrd = <Third>[(Third,Singular,Fem) 'si', (Third,Singular) 'gi', (Third,Plural) 'ili'];
};

suf = 'n';

Accusative accusative = {
    fst = first_suffix(personal.fst, suf);
    snd = personal.snd ^ suf;
    thrd = third_suffix(personal.thrd, suf);
};

pronoun_list = <$PronounDef>[personal, accusative];

typedef $Language = {
    dict<Root,$VerbDef> verbs;
    list<$PronounDef> pronouns;
};

$Language myLang = {
    verbs = verb_dict;
    pronouns = pronoun_list;
};
```

```
print_lang = <$Language lang:int> {
    print_verbs = <Root r, $VerbDef v :$VerbDef>{
        sprint('Root: ' ^ (string)r);
        sprint('Infinitive: ' ^ (string)v.inf);
        sprint('Present: ' ^ (string)v.pres);
        sprint('Past: ' ^ (string)v.past);
        sprint('Future: ' ^ (string)v.fut);
        v;
    };
    dmap(print_verbs, lang.verbs);
    print_pronouns = <$PronounDef p:$PronounDef>{
        sprint('First Person:');
        print_first = <First f : First>{
            sprint((string)f);
            f;
        };
        lmap(print_first, p.fst);

        sprint('Second Person:');
        sprint((string)p.snd);

        sprint('Third Person:');
        print_third = <Third t : Third>{
            sprint((string)t);
            t;
        };
        lmap(print_third, p.thrd);
        p;
    };
    lmap(print_pronouns, lang.pronouns);
    0;
};
print_lang(myLang);
```

14.1 Output

```
Root: est
Infinitive: esti
Present: estas
Past: estis
Future: estos
Root: lern
Infinitive: lerni
Present: lernas
Past: lernis
Future: lernos
Root: hav
Infinitive: havi
Present: havas
Past: havis
Future: havos
First Person:
mi
ni
Second Person:
vi
Third Person:
```

si
gi
ili
First Person:
min
nin
Second Person:
vin
Third Person:
sin
gin
ilin