Concrete Syntax Tree

A library for the manipulation of Common Lisp source code enhanced with information about source origin

This manual is for Concrete Syntax Tree version 0.4.0.

Table of Contents

1 Introduction

Concrete Syntax Tree is a library for manipulating Common Lisp source code enhanced with information about the origin of the source. It is intended to solve a problem that occurs in programs which process Common Lisp source code such as file compilers or sophisticated editors. If such a program processes source code by first calling the Common Lisp `read` function on every top-level expression in a file or buffer, then the information about the origin of those expressions is lost. This loss of information has a serious negative impact on diagnostic messages from the program, because the user does not get direct information about the origin of the code that the respective message refers to. The solution to this problem involves what is called *source tracking*, which basically means that we need to keep track of this origin information, throughout the processing steps. In case of a compiler, source tracking should cover processing steps all the way from source code to executable code.

One requirement for improved source tracking is that the source code must be read by an improved version of the `read` function[1]. A typical solution would be to make `read` keep a hash table that associates the expressions being read to the location in the file. But this solution only works for freshly allocated Common Lisp objects. It will not work for code elements such as numbers, characters, or symbols, simply because there may be several occurrences of similar code elements in the source. The solution provided by this library is to manipulate Common Lisp source code in the form of a *concrete syntax tree*, or CST for short. A CST is simply a wrapper (in the form of a standard instance) around a Common Lisp expression, which makes the representation of every source sub-expression distinct and also provides a place for attaching the additional information required for source tracking. In order to make the manipulation of CSTs as painless as possible for client code, this library provides a set of functions that mimic the ones that would be used on raw Common Lisp code, such as `first`, `rest`, `consp`, `null`, etc.

The exact nature of the origin information in a CST is left to the client. Concrete Syntax Tree just propagates this information as much as possible through the functions in the library that manipulate the source code in the form of CSTs. For example, Concrete Syntax Tree provides code utilities for canonicalizing declarations, parsing lambda lists, separating declarations and documentation strings and code bodies, checking whether a form is a proper list, etc. All these utilities manipulate the code in the form of a CST, and provide CSTs as a result of the manipulation that propagates the origin information as much as possible. In particular, Concrete Syntax Tree provide a helper function for an "intelligent macroexpander": The `concrete-syntax-tree:reconstruct` function takes an original CST and the result of macroexpanding the *raw* code version of that CST, and returns a new CST representing the expanded code in such a way that as much as possible of the origin information is preserved.

[1] The Eclector library (`https://github.com/s-expressionists/Eclector`) provides such a `read` function.

2 User manual

2.1 Basic Use

In this chapter, we describe the basic functionality that Concrete Syntax Tree provides for manipulating concrete syntax trees.

2.1.1 Protocol

`cst` `[concrete-syntax-tree]` [Class]
This class is the base class for all concrete syntax trees.

`:raw` [Initarg]
The value of this initialization argument is the raw Common Lisp expression that this concrete syntax tree represents.

`raw` `[concrete-syntax-tree]` [Generic Function]
cst

This generic function returns the raw Common Lisp expression that is represented by *cst* as provide by the initialization argument `:raw` when *cst* was created.

`:source` [Initarg]
This initialization argument is accepted by all subclasses of concrete syntax trees. The value of this initialization argument is a client-specific object that indicates the origin of the source code represented by this concrete syntax tree. A value of `nil` indicates that the origin of the source code represented by this concrete syntax tree is unknown. The default value (if this initialization argument is not provided) is `nil`.

`source` `[concrete-syntax-tree]` [Generic Function]
cst

This generic function returns the origin information of *cst* as provide by the initialization argument `:source` when *cst* was created.

`null` `[concrete-syntax-tree]` [Generic Function]
cst

This generic function returns *true* if and only if *cst* is an instance of the class `atom-cst` that has `nil` as its raw value. Otherwise, it returns *false*.

`cons-cst` `[concrete-syntax-tree]` [Class]
This class is a subclass of the class `cst`.

`:first` [Initarg]
The value of this initialization argument is the concrete syntax tree that represents the `car` of the raw Common Lisp expression represented by this concrete syntax tree.

`:rest` [Initarg]
The value of this initialization argument is the concrete syntax tree that represents the `cdr` of the raw Common Lisp expression represented by this concrete syntax tree.

`first` `[concrete-syntax-tree]` [Generic Function]
cons-cst

This generic function returns the concrete syntax tree that represents the `car` of the raw Common Lisp expression represented by *cons-cst*.

`rest` `[concrete-syntax-tree]` [Generic Function]
cons-cst

This generic function returns the concrete syntax tree that represents the `cdr` of the raw Common Lisp expression represented by *cons-cst*.

`consp` `[concrete-syntax-tree]` [Generic Function]
cst

This generic function returns *true* if and only if *cst* is an instance of the class `cons-cst`. Otherwise, it returns *false*.

2.1.2 Additional API functions

`second` `[concrete-syntax-tree]` [Generic Function]
cons-cst

`third` `[concrete-syntax-tree]` [Generic Function]
cons-cst

`fourth` `[concrete-syntax-tree]` [Generic Function]
cons-cst

`fifth` `[concrete-syntax-tree]` [Generic Function]
cons-cst

`sixth` `[concrete-syntax-tree]` [Generic Function]
cons-cst

`seventh` `[concrete-syntax-tree]` [Generic Function]
cons-cst

`eighth` `[concrete-syntax-tree]` [Generic Function]
cons-cst

`ninth` `[concrete-syntax-tree]` [Generic Function]
cons-cst

`tenth` `[concrete-syntax-tree]` [Generic Function]
cons-cst

2.2 Parsing lambda lists

The Concrete Syntax Tree library contains a framework for parsing lambda lists. This framework contains functions for parsing the types of lambda lists that specified in the Common Lisp standard, but it also contains a protocol that allows different implementations to specify additional lambda-list keywords, and to specify how lambda lists containing these additional keywords should be parsed.

2.2.1 Classes for grammar symbols

`grammar-symbol` `[concrete-syntax-tree]` [Class]
This class is the root of all grammar-symbol classes.

2.2.1.1 Classes for parameter groups

`parameter-group` `[concrete-syntax-tree]` [Class]
This class is the root class of all classes that represent parameter groups. It is a subclass of the class `grammar-symbol`.

`:children` [Initarg]
This initialization argument can be used with all subclasses of the class named `parameter-group`. For parameter groups that have no lambda-list keywords, such as all the parameter groups corresponding to required parameters, the value of the argument is a (possibly empty) list of parameters. For parameter groups that have associated lambda-list keywords, the value of the argument includes those lambda-list keywords in addition to the parameters themselves.

`singleton-parameter-group-mixin` `[concrete-syntax-tree]` [Class]
This class is used as a superclass of all classes representing parameter groups with a keyword followed by a single parameter.

`:parameter` [Initarg]
This initialization argument can be used with subclasses of the class named `singleton-parameter-group-mixin`, but the parser does not use it. Instead, it passes the `:children` initialization argument. An appropriate `:after` method on `initialize-instance` splits the children into the lambda-list keyword itself and the single following parameter.

`parameter` `[concrete-syntax-tree]` [Generic Function]
singleton-parameter-group-mixin

This generic function returns a concrete syntax tree representing the single parameter of its argument.

`multi-parameter-group-mixin` `[concrete-syntax-tree]` [Class]
This class is used as a superclass of all classes representing parameter groups with or without a keyword followed by a (possibly empty) list of parameters.

`:parameters` [Initarg]
This initialization argument can be used with subclasses of the class named `multi-parameter-group-mixin`, but the parser does not use it. Instead, it passes the `:children` initialization argument. An appropriate `:after` method on `initialize-instance` computes this initialization argument from the children.

`parameters` `[concrete-syntax-tree]` [Generic Function]
multi-parameter-group-mixin

This generic function returns a list of concrete syntax trees representing the parameters of its argument.

`implicit-parameter-group` `[concrete-syntax-tree]` [Class]
This class is the root class of all classes that represent parameter groups that are *not* introduced by a lambda-list keyword, which is all the different classes representing required parameter groups.

This class is a subclass of the class named `parameter-group` and the class named `multi-parameter-group-mixin`.

`explicit-parameter-group` `[concrete-syntax-tree]` [Class]
This class is the root class of all classes that represent parameter groups that *are* introduced by a lambda-list keyword.

This class is a subclass of the class `parameter-group`.

`:keyword` [Initarg]
This initialization argument can be used with subclasses of the class named `explicit-parameter-group`, but the parser does not use it. Instead, it passes the `:children` initialization argument. An appropriate `:after` method on `initialize-instance` computes this initialization argument from the children.

`keyword` `[concrete-syntax-tree]` [Generic Function]
explicit-parameter-group

This generic function returns the lambda-list keyword of its argument.

`explicit-multi-parameter-group` `[concrete-syntax-tree]` [Class]
This class is the root class of all classes that represent parameter groups that are introduced by a lambda-list keyword, and that can take an arbitrary number of parameters.

This class is a subclass of the class named `explicit-parameter-group` and of the class named `multi-parameter-group-mixin`

`ordinary-required-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents the list of required parameters in all lambda lists that only allow a simple variables to represent a required parameter.

This class is a subclass of the class `implicit-parameter-group`.

`optional-parameter-group` `[concrete-syntax-tree]` [Class]
This class is the root class of all classes representing optional parameter groups.

It is a subclass of the class `explicit-multi-parameter-group`.

`ordinary-optional-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents the list of optional parameters in all lambda lists that allow for an optional parameter to have a form representing the default value and a `supplied-p` parameter.

This class is a subclass of the class `optional-parameter-group`.

`key-parameter-group` `[concrete-syntax-tree]` [Class]
This class is the root class of all parameter groups that are introduced by the lambda-list keyword `&key`.

This class is a subclass of the class `explicit-multi-parameter-group`.

`allow-other-keys` `[concrete-syntax-tree]` [Generic Function]
key-parameter-group

This function can be called on any instance of the class `key-parameter-group`. If the lambda-list keyword `\&allow-other-keys` is present in this key parameter group, then `allow-other-keys` returns a concrete syntax tree for that lambda-list keyword. If the lambda-list keyword `\&allow-other-keys` is *not* present, this function returns `nil`.

`ordinary-key-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents the list of `&key` parameters in all lambda lists that allow for the parameter to have a form representing the default value and a `supplied-p` parameter.

This class is a subclass of the class `key-parameter-group`.

`generic-function-key-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents the list of `&key` parameters in a generic-function lambda list. This type of lambda list only allows for the parameter to have a variable name and a keyword.

This class is a subclass of the class `key-parameter-group`.

`aux-parameter-group` `[concrete-syntax-tree]` [Class]
This class represent the list of `&aux` parameters in all lambda lists that allow for such parameters.

This class is a subclass of the class `explicit-multi-parameter-group`.

`generic-function-optional-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents the list of optional parameters in a generic-function lambda list. This type of lambda list only allows for the parameter to have a variable name.

This class is a subclass of the class `optional-parameter-group`.

`specialized-required-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents the list of required parameters in a specialized lambda list, i.e., the type of lambda list that can be present in a `defmethod` definition. In this type of lambda list, a required parameter may optionally have a *specializer* associated with it.

This class is a subclass of the class `implicit-parameter-group`.

`destructuring-required-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents the list of required parameters in a destructuring lambda list, i.e. the kind of lambda list that can be present in a `defmacro` definition, both as the top-level lambda list and as a nested lambda list where this is allowed.

This class is a subclass of the class `implicit-parameter-group`.

`singleton-parameter-group` `[concrete-syntax-tree]` [Class]
This class is the root class of all parameter groups that consist of a lambda-list keyword followed by a single parameter. It is a subclass of the class `explicit-parameter-group`.

This class is a subclass of the class named `explicit-parameter-group` and of the class named `singleton-parameter-group-mixin`.

`ordinary-rest-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents parameter groups that have either the lambda-list keyword `\&rest` or the lambda-list keyword `\&body` followed by a simple-variable.

This class is a subclass of the class `singleton-parameter-group`.

`destructuring-rest-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents parameter groups that have either the lambda-list keyword `\&rest` or the lambda-list keyword `\&body` followed by a destructuring parameter.

This class is a subclass of the class `singleton-parameter-group`.

`environment-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents parameter groups that have the lambda-list keyword `\&environment` followed by a simple-variable.

This class is a subclass of the class `singleton-parameter-group`.

`whole-parameter-group` `[concrete-syntax-tree]` [Class]
This class represents parameter groups that have the lambda-list keyword `\&whole` followed by a simple-variable.

This class is a subclass of the class `singleton-parameter-group`.

2.2.1.2 Classes for individual parameters

`name` `[concrete-syntax-tree]` [Generic Function]
parameter

`parameter` `[concrete-syntax-tree]` [Class]
This class is the root class of all classes that represent individual lambda-list parameters.

This class is a subclass of the class `grammar-symbol`.

`form-mixin` `[concrete-syntax-tree]` [Class]
This mixin class is a superclass of subclasses of `parameter` that allow for an optional form to be supplied, which will be evaluated to supply a value for a parameter that is not explicitly passed as an argument.

`:form` [Initarg]
This initialization argument can be used when an instance of (a subclass of) the class `form-mixin` is created. The value of this initialization argument is either a concrete syntax tree representing the form that was supplied in the lambda list, or `nil`, indicating that no form was supplied.

`form` `[concrete-syntax-tree]` [Generic Function]
form-mixin

This generic function can be applied to instances of all subclasses of the class `form-mixin`. It returns the value that was supplied using the `:form` initialization argument when the instance was created.

`supplied-p-mixin` `[concrete-syntax-tree]` [Class]
This mixin class is a superclass of subclasses of `parameter` that allow for an optional supplied-p parameter to be supplied. At run-time, the value of this parameter indicates whether an explicit argument was supplied that provides a value for the parameter.

`:supplied-p` [Initarg]
This initialization argument can be used when an instance of (a subclass of) the class `supplied-p-mixin` is created. The value of this initialization argument is either a concrete syntax tree representing the supplied-p parameter that was given in the lambda list, or `nil`, indicating that no supplied-p parameter was given.

`supplied-p` `[concrete-syntax-tree]` [Generic Function]
supplied-p-mixin

This generic function can be applied to instances of all subclasses of the class `supplied-p-mixin`. It returns the value that was provided using the `:supplied-p` initialization argument when the instance was created.

`keyword-mixin` `[concrete-syntax-tree]` [Class]
This mixin class is a superclass of subclasses of `parameter` that allow for an optional keyword parameter to be supplied.

`:keyword` [*keyword-mixin*] [Initarg]
This initialization argument can be used when an instance of (a subclass of) the class `keyword-mixin` is created. The value of this initialization argument is either a concrete syntax tree representing the keyword parameter that was given in the lambda list, or `nil`, indicating that no keyword parameter was given.

The generic function [Generic-Function concrete-syntax-tree|keyword], page 5, can be applied to instances of all subclasses of the class `keyword-mixin`. It returns the value that was provided using the `:keyword` initialization argument when the instance was created.

`simple-variable` `[concrete-syntax-tree]` [Class]
This class represents lambda-list parameters that must be simple variables, for example the required parameters in an ordinary lambda list, or the parameter following the `\&environment` lambda-list keyword.

This class is a subclass of the class `parameter`.

`ordinary-optional-parameter` `[concrete-syntax-tree]` [Class]
This class represents an optional parameter of the form that is allowed in an ordinary lambda list, i.e., a parameter that, in addition to the parameter name, can have a form representing the default value and an associated `supplied-p` parameter.

This class is a subclass of the class named `parameter`, the class named `form-mixin`, and the class named `supplied-p-mixin`.

`ordinary-key-parameter` `[concrete-syntax-tree]` [Class]
This class represents a `&key` parameter of the form that is allowed in an ordinary lambda list, i.e., a parameter that, in addition to the parameter name, can have

a keyword, a form representing the default value, and an associated `supplied-p` parameter.

This class is a subclass of the class named `parameter`, the class named `form-mixin`, the class named `supplied-p-mixin`, and the class named `keyword-mixin`.

`generic-function-key-parameter` `[concrete-syntax-tree]` [Class]
This class represents a `&key` parameter of the form that is allowed in a generic-function lambda list, i.e., a parameter that, in addition to the parameter name, can have a keyword, but no form representing the default value, and no associated `supplied-p` parameter.

This class is a subclass of the class named `parameter`, and the class named `keyword-mixin`.

`aux-parameter` `[concrete-syntax-tree]` [Class]
This class represents an `&aux` parameter, i.e., a parameter that, in addition to the parameter name, can have a form representing the default value.

This class is a subclass of the classes `parameter` and `form-mixin`.

`generic-function-optional-parameter` `[concrete-syntax-tree]` [Class]
This class represents an optional parameter of the form that is allowed in an ordinary lambda list, i.e., a parameter that can only have a name, but that name can optionally be the element of a singleton list, which is why it is distinct from a parameter of type `simple-variable`.

This class is a subclass of the class `parameter`.

`specialized-required-parameter` `[concrete-syntax-tree]` [Class]
This class represents the kind of required parameter that can appear in a specialized lambda list, i.e. a lambda list in a `defmethod` form. A parameter of this type may optionally have a *specializer* in the form of a class name or an `eql` specializer associated with it.

This class is a subclass of the class `parameter`.

`destructuring-parameter` `[concrete-syntax-tree]` [Class]
This class represents all parameters that can be either simple variables or a pattern in the form of a (possibly dotted) list, or even a destructuring lambda list.

This class is a subclass of the class `grammar-symbol`.

2.2.1.3 Classes for lambda-list keywords

`lambda-list-keyword` `[concrete-syntax-tree]` [Class]
This class is the root class of all classes representing lambda-list keywords.

This class is a subclass of the class `grammar-symbol`.

The generic function [Generic-Function concrete-syntax-tree|name], page 7, is applicable to every instantiable subclass of the class `lambda-list-keyword`. It returns the concrete syntax tree corresponding to the lambda-list keyword in the original lambda list.

`keyword-optional` `[concrete-syntax-tree]` [Class]
This class represents the lambda-list keyword `\&optional`. Unless client code has defined some additional lambda-list keyword that is used in the same way as `\&optional`, an instance of this class always represents the lambda-list keyword `\&optional`.

This class is a subclass of the class `lambda-list-keyword`.

`keyword-rest` `[concrete-syntax-tree]` [Class]
This class represents the lambda-list keyword `\&rest`. Unless client code has defined some additional lambda-list keyword that is used in the same way as `\&rest`, an instance of this class always represents the lambda-list keyword `\&rest`.

This class is a subclass of the class `lambda-list-keyword`.

`keyword-body` `[concrete-syntax-tree]` [Class]
This class represents the lambda-list keyword `\&body`. Unless client code has defined some additional lambda-list keyword that is used in the same way as `\&body`, an instance of this class always represents the lambda-list keyword `\&body`.

This class is a subclass of the class `lambda-list-keyword`.

`keyword-key` `[concrete-syntax-tree]` [Class]
This class represents the lambda-list keyword `&key`. Unless client code has defined some additional lambda-list keyword that is used in the same way as `&key`, an instance of this class always represents the lambda-list keyword `&key`.

This class is a subclass of the class `lambda-list-keyword`.

`keyword-allow-other-keys` `[concrete-syntax-tree]` [Class]
This class represents the lambda-list keyword `\&allow-other-keys`. Unless client code has defined some additional lambda-list keyword that is used in the same way as `\&allow-other-keys`, an instance of this class always represents the lambda-list keyword `\&allow-other-keys`.

This class is a subclass of the class `lambda-list-keyword`.

`keyword-aux` `[concrete-syntax-tree]` [Class]
This class represents the lambda-list keyword `&aux`. Unless client code has defined some additional lambda-list keyword that is used in the same way as `&aux`, an instance of this class always represents the lambda-list keyword `&aux`.

This class is a subclass of the class `lambda-list-keyword`.

`keyword-environment` `[concrete-syntax-tree]` [Class]
This class represents the lambda-list keyword `\&environment`. Unless client code has defined some additional lambda-list keyword that is used in the same way as `\&environment`, an instance of this class always represents the lambda-list keyword `\&environment`.

This class is a subclass of the class `lambda-list-keyword`.

`keyword-whole` `[concrete-syntax-tree]` [Class]
This class represents the lambda-list keyword `\&whole`. Unless client code has defined some additional lambda-list keyword that is used in the same way as `\&whole`, an instance of this class always represents the lambda-list keyword `\&whole`.

This class is a subclass of the class `lambda-list-keyword`.

2.2.1.4 Classes for entire lambda lists

Notice that there is no class for the boa lambda list, since it is syntactically equivalent to an ordinary lambda list. Similarly, there is no `deftype` lambda list because it is syntactically equivalent to a macro lambda list.

`lambda-list-type` `[concrete-syntax-tree]` [Class]

This class is a subclass of the class `grammar-symbol`.

`children` `[concrete-syntax-tree]` [Generic Function]

lambda-list-type

This generic function returns a (possibly empty) list of instances of (some subclass of) the class `parameter-group` as described in Section 2.2.1.1 [Classes for parameter groups], page 4.

`ordinary-lambda-list` `[concrete-syntax-tree]` [Class]

Applying the function `children` to an instance of this class returns a list with the following elements (in that order):

1. A mandatory instance of the grammar-symbol class named `ordinary-required-parameter-group`.
2. An optional instance of the grammar-symbol class named `ordinary-optional-parameter-group`.
3. An optional instance of the grammar-symbol class named `ordinary-rest-parameter-group`.
4. An optional instance of the grammar-symbol class named `ordinary-key-parameter-group`.
5. An optional instance of the grammar-symbol class named `aux-parameter-group`.

This class is a subclass of the class `lambda-list-type`.

`generic-function-lambda-list` `[concrete-syntax-tree]` [Class]

Applying the function `children` to an instance of this class returns a list with the following elements (in that order):

1. A mandatory instance of the grammar-symbol class named `ordinary-required-parameter-group`.
2. An optional instance of the grammar-symbol class named `generic-function-optional-parameter`
3. An optional instance of the grammar-symbol class named `ordinary-rest-parameter-group`.
4. An optional instance of the grammar-symbol class named `generic-function-key-parameter-group`
5. An optional instance of the grammar-symbol class named `aux-parameter-group`.

This class is a subclass of the class `lambda-list-type`.

`specialized-lambda-list` `[concrete-syntax-tree]` [Class]

Applying the function `children` to an instance of this class returns a list with the following elements (in that order):

1. A mandatory instance of the grammar-symbol class named `specialized-required-parameter-gro`
2. An optional instance of the grammar-symbol class named `ordinary-optional-parameter-group`.
3. An optional instance of the grammar-symbol class named `ordinary-rest-parameter-group`.
4. An optional instance of the grammar-symbol class named `ordinary-key-parameter-group`.
5. An optional instance of the grammar-symbol class named `aux-parameter-group`.

This class is a subclass of the class `lambda-list-type`.

`defsetf-lambda-list` `[concrete-syntax-tree]` [Class]

Applying the function `children` to an instance of this class returns a list with the following elements (in that order):

1. A mandatory instance of the grammar-symbol class named `ordinary-required-parameter-group`.
2. An optional instance of the grammar-symbol class named `ordinary-optional-parameter-group`.
3. An optional instance of the grammar-symbol class named `ordinary-rest-parameter-group`.
4. An optional instance of the grammar-symbol class named `t-key-parameter-group`.
5. An optional instance of the grammar-symbol class named `environment-parameter-group`.

This class is a subclass of the class `lambda-list-type`.

`define-modify-macro-lambda-list` `[concrete-syntax-tree]` [Class]

Applying the function `children` to an instance of this class returns a list with the following elements (in that order):

1. A mandatory instance of the grammar-symbol class named `ordinary-required-parameter-group`.
2. An optional instance of the grammar-symbol class named `ordinary-optional-parameter-group`.
3. An optional instance of the grammar-symbol class named `ordinary-rest-parameter-group`.

This class is a subclass of the class `lambda-list-type`.

`define-method-combination-lambda-list` `[concrete-syntax-tree]` [Class]

Applying the function `children` to an instance of this class returns a list with the following elements (in that order):

1. An optional instance of the grammar-symbol class named `whole-parameter-group`.
2. A mandatory instance of the grammar-symbol class named `ordinary-required-parameter-group`.
3. An optional instance of the grammar-symbol class named `ordinary-optional-parameter-group`.
4. An optional instance of the grammar-symbol class named `ordinary-rest-parameter-group`.
5. An optional instance of the grammar-symbol class named `ordinary-key-parameter-group`.
6. An optional instance of the grammar-symbol class named `aux-parameter-group`.

This class is a subclass of the class `lambda-list-type`.

`macro-lambda-list` `[concrete-syntax-tree]` [Class]

This class is a subclass of the class `lambda-list-type`.

`destructuring-lambda-list` `[concrete-syntax-tree]` [Class]

This class is a subclass of the class `lambda-list-type`.

`target` `[concrete-syntax-tree]` [Class]

This class is a subclass of the class `grammar-symbol`.

2.2.2 Variables

2.2.2.1 Parameter groups

`*ordinary-required-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `ordinary-required-parameter-group`. This rule defines an ordinary required parameter group as a (possibly empty) sequence of instances of the class `simple-variable`.

`*ordinary-optional-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `ordinary-optional-parameter-group`. This rule defines an ordinary optional parameter group as the lambda list keyword `\&optional`, followed by a (possibly empty) sequence of instances of the class `ordinary-optional-parameter`.

`*ordinary-rest-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `ordinary-rest-parameter-group`. This rule defines an ordinary rest parameter group as the lambda list keyword `\&rest`, followed by an instances of the class `simple-variable`.

`*ordinary-key-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `ordinary-key-parameter-group`. This rule defines an ordinary key parameter group as the lambda list keyword `&key`, followed by a (possibly empty) sequence of instances of the class `ordinary-key-parameter`, optionally followed by the lambda-list keyword `\&allow-other-keys`.

`*aux-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `aux-parameter-group`. This rule defines an aux parameter group as the lambda list keyword `&aux`, followed by a (possibly empty) sequence of instances of the class `aux-parameter`.

`*generic-function-optional-parameter-group*` [Variable]
`[concrete-syntax-tree]`
This variable defines a grammar rule for the grammar-symbol class named `generic-function-optional-parameter-group`. This rule defines a generic-function optional parameter group as the lambda list keyword `\&optional`, followed by a (possibly empty) sequence of instances of the class named `generic-function-optional-parameter`.

`*generic-function-key-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `generic-function-key-parameter-group`. This rule defines an generic-function key parameter group as the lambda list keyword `&key`, followed by a (possibly empty) sequence of instances of the class `generic-function-key-parameter`, optionally followed by the lambda-list keyword `\&allow-other-keys`.

`*specialized-required-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `specialized-required-parameter-group`. This rule defines an specialized required parameter group as a (possibly empty) sequence of instances of the class `specialized-required-parameter`.

`*environment-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `environment-parameter-group`. This rule defines an environment parameter group as the lambda list keyword `\&environment`, followed by an instances of the class `simple-variable`.

`*whole-parameter-group*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `whole-parameter-group`. This rule defines a whole parameter group as the lambda list keyword `\&whole`, followed by an instances of the class `simple-variable`.

`*destructuring-required-parameter-group*,concrete-syntax-tree` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `destructuring-required-parameter-group`. This rule defines an destructuring required parameter group as a (possibly empty) sequence of instances of the class `destructuring-parameter`.

`*destructuring-rest-parameter-group*,concrete-syntax-tree` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `destructuring-rest-parameter-group`. This rule defines a destructuring rest parameter group as the lambda list keyword `\&rest`, followed by an instances of the class `destructuring-parameter`.

2.2.2.2 Lambda-list types

`*ordinary-lambda-list*` `[concrete-syntax-tree]` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `ordinary-lambda-list`. It has the following definition:

```
(defparameter *ordinary-lambda-list*
  '((ordinary-lambda-list <-
     ordinary-required-parameter-group
     (? ordinary-optional-parameter-group)
     (? ordinary-rest-parameter-group)
     (? ordinary-key-parameter-group)
     (? aux-parameter-group))))
```

`*generic-function-lambda-list*,concrete-syntax-tree` [Variable]
This variable defines a grammar rule for the grammar-symbol class named `generic-function-lambda-list`. It has the following definition:

```
(defparameter *generic-function-lambda-list*
  '((generic-function-lambda-list <-
     ordinary-required-parameter-group
```

```
(? generic-function-optional-parameter-group)
(? ordinary-rest-parameter-group)
(? generic-function-key-parameter-group))))
```

`*specialized-lambda-list*` `[concrete-syntax-tree]` [Variable]

This variable defines a grammar rule for the grammar-symbol class named `specialized-lambda-list`. It has the following definition:

```
(defparameter *specialized-lambda-list*
  '((specialized-lambda-list <-
     specialized-required-parameter-group
     (? ordinary-optional-parameter-group)
     (? ordinary-rest-parameter-group)
     (? ordinary-key-parameter-group)
     (? aux-parameter-group))))
```

`*defsetf-lambda-list*` `[concrete-syntax-tree]` [Variable]

This variable defines a grammar rule for the grammar-symbol class named `defsetf-lambda-list`. It has the following definition:

```
(defparameter *defsetf-lambda-list*
  '((defsetf-lambda-list <-
     ordinary-required-parameter-group
     (? ordinary-optional-parameter-group)
     (? ordinary-rest-parameter-group)
     (? ordinary-key-parameter-group)
     (? environment-parameter-group))))
```

`*define-modify-macro-lambda-list*,concrete-syntax-tree` [Variable]

This variable defines a grammar rule for the grammar-symbol class named `define-modify-macro-lambda-list`. It has the following definition:

```
(defparameter *define-modify-macro-lambda-list*
  '((define-modify-macro-lambda-list <-
     ordinary-required-parameter-group
     (? ordinary-optional-parameter-group)
     (? ordinary-rest-parameter-group))))
```

`*define-method-combination-lambda-list*` `[concrete-syntax-tree]` [Variable]

This variable defines a grammar rule for the grammar-symbol class named `define-method-combination-lambda-list`. It has the following definition:

```
(defparameter *define-method-combination-lambda-list*
  '((define-method-combination-lambda-list <-
     (? whole-parameter-group)
     ordinary-required-parameter-group
     (? ordinary-optional-parameter-group)
     (? ordinary-rest-parameter-group)
     (? ordinary-key-parameter-group)
     (? aux-parameter-group))))
```

`*destructuring-lambda-list*` `[concrete-syntax-tree]` [Variable]

This variable defines a grammar rule for the grammar-symbol class named `destructuring-lambda-list`.

This rule defines a destructuring lambda list as sequence of the following items:

1. An optional instance of the grammar-symbol class named `whole-parameter-group`.
2. A mandatory instance of the grammar-symbol class named `destructuring-required-parameter-g`
3. An optional instance of the grammar-symbol class named `ordinary-optional-parameter-group`.
4. An optional instance of the grammar-symbol class named `destructuring-rest-parameter-group`.
5. An optional instance of the grammar-symbol class named `ordinary-key-parameter-group`.
6. An optional instance of the grammar-symbol class named `aux-parameter-group`.

`*macro-lambda-list*` `[concrete-syntax-tree]` [Variable]

This variable defines a grammar rule for the grammar-symbol class named `macro-lambda-list`.

This rule defines a macro lambda list as sequence of the following items:

1. An optional instance of the grammar-symbol class named `whole-parameter-group`.
2. An optional instance of the grammar-symbol class named `environment-parameter-group`.
3. A mandatory instance of the grammar-symbol class named `destructuring-required-parameter-g`
4. An optional instance of the grammar-symbol class named `environment-parameter-group`.
5. An optional instance of the grammar-symbol class named `ordinary-optional-parameter-group`.
6. An optional instance of the grammar-symbol class named `environment-parameter-group`.
7. An optional instance of the grammar-symbol class named `destructuring-rest-parameter-group`.
8. An optional instance of the grammar-symbol class named `environment-parameter-group`.
9. An optional instance of the grammar-symbol class named `ordinary-key-parameter-group`.
10. An optional instance of the grammar-symbol class named `environment-parameter-group`.
11. An optional instance of the grammar-symbol class named `aux-parameter-group`.
12. An optional instance of the grammar-symbol class named `environment-parameter-group`.

Notice that this definition allows for there to be several occurrences of the grammar symbol `environment-parameter-group`, whereas the Common Lisp standard allows for at most one such occurrence. The top-level parser for this type of lambda list checks that at most one such occurrence is present after parsing is complete.

2.2.2.3 Full grammars

In order to have a grammar that is possible to use for parsing a lambda list, in addition to the rules for all the lambda list types and its components, a target rule is required to initialize a grammar object with `generate-grammar`, which the parser ultimately uses. The target is in accordance to the particular lambda-list type that is desired.

`*standard-grammar*` `[concrete-syntax-tree]` [Variable]

This variable contains all the standard grammar rules in description form.

`*ordinary-lambda-list-grammar*` `[concrete-syntax-tree]` [Variable]

This variable contains a grammar object with all the standard grammar rules, with target `ordinary-lambda-list-grammar`.

`*generic-function-lambda-list-grammar*` `[concrete-syntax-tree]` [Variable]
This variable contains a grammar object with all the standard grammar rules, with target `generic-function-lambda-list`.

`*specialized-lambda-list-grammar*` `[concrete-syntax-tree]` [Variable]
This variable contains a grammar object with all the standard grammar rules, with target `specialized-lambda-list`.

`*defsetf-lambda-list-grammar*` `[concrete-syntax-tree]` [Variable]
This variable contains a grammar object with all the standard grammar rules, with target `defsetf-lambda-list`.

`*define-modify-macro-lambda-list-grammar*` `[concrete-syntax-tree]` [Variable]
This variable contains a grammar object with all the standard grammar rules, with target `define-modify-macro-lambda-list`.

`*define-method-combination-lambda-list-grammar*` [Variable]
`[concrete-syntax-tree]`
This variable contains a grammar object with all the standard grammar rules, with target `define-method-combination-lambda-list`.

`*destructuring-lambda-list-grammar*` `[concrete-syntax-tree]` [Variable]
This variable contains a grammar object with all the standard grammar rules, with target `destructuring-lambda-list`.

`*macro-lambda-list-grammar*` `[concrete-syntax-tree]` [Variable]
This variable contains a grammar object with all the standard grammar rules, with target `macro-lambda-list`.

2.2.3 Parsers for standard lambda lists

`parse-ordinary-lambda-list` `[concrete-syntax-tree]` [Generic Function]
client lambda-list (error-p t)

`parse-generic-function-lambda-list` `[concrete-syntax-tree]` [Generic Function]
client lambda-list (error-p t)

`parse-specialized-lambda-list` `[concrete-syntax-tree]` [Generic Function]
client lambda-list (error-p t)

`parse-defsetf-lambda-list` `[concrete-syntax-tree]` [Generic Function]
client lambda-list (error-p t)

`parse-define-modify-macro-lambda-list` [Generic Function]
`[concrete-syntax-tree]`
client lambda-list (error-p t)

`parse-define-method-combination-lambda-list` [Generic Function]
`[concrete-syntax-tree]`
client lambda-list (error-p t)

`parse-destructuring-lambda-list` `[concrete-syntax-tree]` [Generic Function]
client lambda-list (error-p t)

`parse-macro-lambda-list` `[concrete-syntax-tree]` [Generic Function]
client lambda-list (error-p t)

2.3 Destructuring lambda lists

When applied to lambda lists, the term *destructuring* means to match its parameters against an argument list, and to generate a set of nested `let` bindings. A binding will bind a parameter of the lambda list to its corresponding value in the argument list, or it will bind some temporary variable. The argument list is not known at the time of the destructuring, so the *form* of each binding will consist of calls to destructuring functions such as `car` and `cdr`, starting with a variable that holds the entire argument list as its value.

This kind of destructuring is used at macro-expansion time when certain macros are expanded. In particular `defmacro` and `define-compiler-macro`. The result of the destructuring is a `lambda` expression for the *macro function*. This lambda expression is then compiled to create the final macro function.

Every function defined here wraps a *body* form in some `let` bindings. These `let` bindings are determined by the parameters of a lambda list. Each function handles a different part of the lambda list. The *client* parameter is some object representing the client. It is used among other things to determine which condition class to use when a a condition needs to be signaled. The *argument-variable* parameter (abbreviated *av* is a symbol that, when the resulting macro function is executed on some compound form corresponding to a macro call, will hold the remaining part of the arguments of that macro call yet to be processed.

Some functions have an argument called *tail-variable* (abbreviated *tv*), which is also a symbol that is going to be used in subsequent destructuring functions for the same purpose as *argument-variable*. Such a function is responsible for creating an innermost `let` form that binds the *tail-variable* symbol to the part of the argument list that remains after the function has done its processing. Some functions do not need such a variable, because they do not consume any arguments, so the remaining argument list is the same as the initial one.

`destructure-lambda-list` `[concrete-syntax-tree]` [Generic Function]
client lambda-list av tv body

Given an entire lambda list, which can be a macro lambda list or a destructuring lambda list, wrap *body* in a bunch of nested `let` bindings according to the parameters of the lambda list.

`destructure-aux-parameter` `[concrete-syntax-tree]` [Generic Function]
client parameter body

Wrap *body* in a `let` form corresponding to a single `aux` parameter. Since `aux` parameters are independent of the macro-call arguments, there is no need for an `argument-variable`. The `aux` parameter itself provides all the information required to determine the `let` binding.

`destructure-aux-parameters` `[concrete-syntax-tree]` [Generic Function]
client parameters body

Wrap *body* in nested `let` forms, each corresponding to a single `aux` parameter in the list of `aux` parameters *parameters*. Since `aux` parameters are independent of the macro-call argument, there is no need for an *argument-variable*. Each `aux` parameter in *parameters* itself provides all the information required to determine the `let` binding.

`destructure-key-parameter` `[concrete-syntax-tree]` [Generic Function]
client parameter av body

Wrap *body* in a `let` form corresponding to a single `key` parameter.

`destructure-key-parameters` `[concrete-syntax-tree]` [Generic Function]
client parameters av body

Wrap *body* in nested `let` forms, each corresponding to a single `key` parameter in a list of such `key` parameters. Since `key` parameters do not consume any arguments, the list of arguments is the same before and after the `key` parameters have been processed. As a consequence, we do not need a *tail-variable* for `key` parameters.

`destructure-rest-parameter` `[concrete-syntax-tree]` [Generic Function]
client parameter av body

Wrap *body* in a `let` form corresponding to a `rest` parameter. Since `rest` parameters do not consume any arguments, the list of arguments is the same before and after the `rest` parameter has been processed. As a consequence, we do not need a *tail-variable* for `rest` parameters.

`destructure-optional-parameter` `[concrete-syntax-tree]` [Generic Function]
client parameter av body

Wrap *body* in a `let` form corresponding to a single `optional` parameter.

`destructure-optional-parameters` `[concrete-syntax-tree]` [Generic Function]
client parameters av tv body

Wrap *body* in nested `let` forms, each corresponding to a single `optional` parameter in a list of such `optional` parameters. Since every `optional` parameter *does* consume an argument, this function does take a `tail-variable` argument as described above.

`destructure-required-parameter` `[concrete-syntax-tree]` [Generic Function]
client parameter av body

Wrap *body* in one or more `let` forms corresponding to a single required parameter, depending on whether the required parameter is a simple variable or a destructuring lambda list.

`destructure-required-parameters` `[concrete-syntax-tree]` [Generic Function]
client parameters av tv body

Wrap *body* in nested `let` forms, corresponding to the list of required parameters in the list of required parameters *parameters*. Since every required parameter *does* consume an argument, this function does take a `tail-variable` argument as described above.

`destructure-parameter-group` `[concrete-syntax-tree]` [Generic Function]
client group av tv body

Wrap *body* in nested `let` forms, corresponding to the parameters in the list of parameter groups `parameter-groups`.

2.4 Future additions to this library

3 Internals

3.1 Lambda-list Parsing

For parsing lambda lists, we use a technique invented by Jay Earley in 1970 (*Earley, J. (1970). An Efficient Context-free Parsing Algorithm. Commun. ACM, 13(2), 94–102.*, *Earley, J. (1983). An Efficient Context-free Parsing Algorithm. Commun. ACM, 26(1), 57–61.*).

Concept index

(Index is nonexistent)

Function and macro and variable and type index

*

:

A

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

W

Changelog

Release 0.4 (not yet released)

Release 0.3 (2025-06-13)

- A malformed `LOOP` in the internal macro `concrete-syntax-tree:with-bounded-recursion` has been fixed. Most implementations accepted the malformed loop and evaluated it with the intended semantics but Clasp is more strict and therefore required this fix.
- The manual has been converted from LaTeX to texinfo.
- An automatically generated Changelog section has been added to manual.

Release 0.2 (2025-06-07)

- concrete-syntax-tree now uses the fiveam system for its unit tests.
- `concrete-syntax-tree:cst-from-expression` can now handle arbitrary nesting depths and list lengths.
- `concrete-syntax-tree:reconstruct` can now handle arbitrary nesting depths and list lengths.
- `concrete-syntax-tree:cst-from-expression` now creates distinct CSTs for certain atoms that occur multiple times in the source expression, namely numbers, characters and symbols. For other atoms such as pathnames or instances of standard classes, `eq` occurrences in the source expression are represented by multiple occurrences of a single CST.

  For example, the source expression `(1 1 :foo :foo #1=#P"foo" #1#)` was represented before this change as a CST with the following properties

  ```
  (eq (cst:first result) (cst:second result))
  (eq (cst:third result) (cst:fourth result))
  (eq (cst:fifth result) (cst:sixth result))
  ```

  but is now represented as a CST with the following properties

  ```
  (not (eq (cst:first result) (cst:second result)))
  (not (eq (cst:third result) (cst:fourth result)))
  (eq (cst:fourth result) (cst:sixth result))
  ```

- `concrete-syntax-tree:cst-from-expression` and `concrete-syntax-tree:reconstruct` are now slightly more efficient.

Release 0.1 (2023-03-16)

- Initial version with CST classes, `concrete-syntax-tree:cst-from-expression`, `concrete-syntax-tree:reconstruct`, lambda list parsing, and a LaTeX-based manual.