1 Introduction

The XE programming environment [1] includes a compiler for the XE language [2] with two code generators, a database for XE source objects, a ZMACS editor customized for the XE language and an error handling facility. In addition to these main modules there are auxiliary facilities for the generation of compiler data structures etc., for browsing the database, for profiling purposes, and for debugging purposes... in all the XE programming environment is a large and complicated program. Before attempting to modify this environment in any way a throughout reading of documents [2] [1] [4] is recommended.

The purpose of this manual is to serve as a reference guide to the XE environment for those who need to understand how it has been implemented. The manual begins with a short introduction to the basic concepts of the XE environment, then the operational structure is briefly explained. The body of this manual consists of a detailed description of the source modules, which make up the XE environment. The environment is implemented using the Common Lisp [6] language in a Symbolics computer.

2 Top level operation

It is impossible to describe the operation of the XE programming environment without using the concepts of XE language so a short introduction to its basic structures is needed.

2.1 XE language modules and their organization

XE [2] is a data abstraction language descending from Clu [3], without being its direct extension. An XE program is made up of one or more program units also called modules. Each program unit is a *procedure*, *iterator*, *rule*, *data type*, or *constant*.

Program units can have a nested structure relative to one another. For example, it is possible to nest a procedure definition inside another one to obtain an internal procedure. XE has no global variables, but variables and arguments of the surrounding unit can be bound by the nested unit. Thus it is possible to write lexical closures in XE.

All program units, except constants, can be parameterized. This allows a set of related abstractions to be defined by a single unit. A parameterized program unit is called a *generator*, because it defines an ordinary program unit each time it is provided with legal actual parameters; for example `array[int]` and `array[string]`.

XE program units can also be parameterized by *field lists*, i.e., lists of name-type pairs, such as `record{name: string, age: int}`.

Each program unit has a *header* that specifies the interface (signature) of the unit and a *body* containing the implementation. The implementation is not visible to other program units. As in Clu, no special interface or specification units are used.

The elements of the set of implementations of a module are called *revisions*. The set can be partitioned into several *versions*. The versions differ from each other conceptually. They either perform a different task or the same task in a different way. The revisions of a version reflect the development of that version. They are the units of code that the programmer edits and that can be executed. Every revision has a (possibly different) *header*. The structure of the header depends on the syntactic domain of the module. For example, the header of a procedure consists of the argument types, the return types, the signals and the where clauses of the procedure.

A *system* is the topmost hierarchical component manipulated by the programmer in the XE programming environment [1]. For example, a program or a set of programs that are used to perform a task can be a system. Another example is a graphic package that contains type definitions of various graphical entities and routines that manipulate them.

There can be many *generations* of the same system. The generations correspond to the historical development of the system and they form a tree. A generation associates with every module name that it uses a single implementation (some revision of that module). These associations are changed either manually or automatically as the system evolves.

Often it is necessary to use modules of another system in a system. To achieve this it is possible to *export* and *import* modules. A module is imported to the system if it is referred to in it, but not defined in it. Only modules, which are exported can be imported and only from systems which are *included* into the system, which refers to them. If the module could be imported from several systems *a name conflict* is said to arise and the user must resolve which binding he wants to use. The restrictions specified above to the import mechanism creatly reduce the amount of name conflicts.

As module names are associated to their implementations in generations of systems, the inclusion is also done on this level; i.e., a system generation includes specific generations of the systems.

2.2 Compilation of XE Programs

The phase structure of the XE compiler is conventional. The front end consists of

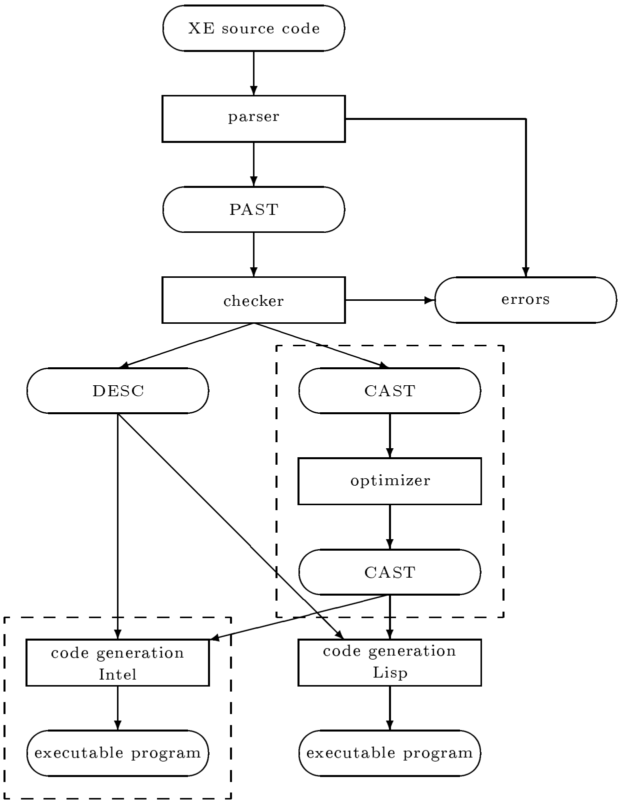

Figure 1: Information flow in XE compiler.

a scanner, a parser that builds an explicit parse tree called *PAST*, and a checker that verifies that context conditions are satisfied and updates the compilation environment by building abstract syntax tree structures called *CAST* and *descriptors* 1. Currently the compiler has two back ends, one that produces Lisp code [7] and another that produces Amsterdam Compiler Kit intermediate code EM [5].

The compile-time information of a program unit is stored in a descriptor. The descriptor contains also the parse tree of the unit and compilation dependencies, i.e., a list of units that this unit refers to, and a list of units that refer to this unit. The descriptors are stored in a symbol table. The symbol table and the descriptors form a persistent compilation environment that remains in the virtual memory during a program development session and that is updated by each compilation.

The XE environment supports separate compilation of program units. Nested subunits, e.g., routines of a data type or local procedures are also compiled separately. The compilation of a program unit is not independent of the compilation of other units because program units refer to each other and the legality of these references must be checked.

The control of XE compiler is also quite complex. It is based on the idea of demand driven *lazy* computations. No compilation order is computed before the compilation. Instead, the compiler starts to compile an arbitrary program unit and when a reference to another unit, that is not already compiled, is detected the compilation of the first unit is suspended and the compilation of the second unit is started. When that compilation has completed the compilation of the first unit is resumed. In order to avoid deadlocks in case of recursive dependencies, all parts of the second unit cannot always be compiled. Only those parts that are needed in the compilation of the first unit are compiled.

The XE compiler also incorporates a *smart recompilation algorithm*, which determines the set of program units that are affected when some program units are changed. The algorithm is phase oriented, which means that for all affected program units it is known what phases of the compiler need to be run in a recompilation. No compilation order needs to be explicitly determined for the affected program units because the demand driven compilation method is used also in recompilations.

2.3 Structure of the Environment

The module structure of XE programming environment is pictured in 2. The main modules are the *database*, the *ZMACS editor*, and the XE compiler. The database is used to store the source modules, and information of system and generation structure. The ZMACS editor is used both as an editor and as a command processor. An the XE compiler is responsible of the transformation of XE sources to Lisp and EM code. The compiler consists of parser, checker, and two code generators as explained in the previous chapter.

The ability to retain persistent data is central in the XE programming environment. The information generated by the XE compiler is kept in the data structures of the environment and is accessible throughout the environment.

The data has several representations. In the database the source code of modules is stored as *text files* and the information of the development of the generations and the system as a *history file*. In the ZMACS editor the programs are represented as ZMACS buffers. The data structure generated by the parser is an abstract syntax tree structure called *PAST*. The checker phase of the compiler checks the abstract syntax trees generated by the parser and creates the checker abstract syntax tree structures called *CAST*. The different syntactical entities of the XE language are represented as *descriptors*. The descriptors contain all the information necessary to manipulate the entities.

Some of the information is kept for compiler control purposes. The error structures are stored into the error base and can be examined using the *error examiner* or the ZMACS editor. The dependency information necessary to control the compilation

5

Figure 2: The structure of the XE compiler

order is kept in the descriptors and used by the *compiler control system.* The descriptor ready for code generation are queued into *code generation queue.* But the most important information is stored into descriptors and kept in a permanent *symbol table.*

The descriptors in symbol table are usually referred to by using the scoping mechanism of the XE compiler. The scoping mechanism is quite complex. There are four different types of names *basic names, signals, handlers,* and *iterations*, which all obey scopes of their own. As the system allows the simultaneous development of several generations the names of top level objects are unique only within a generation. This means that the descriptors are searched using both their name, type, scope id, and generation. The visibility of objects through scope borders depends on their type and the type of the scopes. Often a scope can be deleted and all symbols stored in it can be released when the compiler closes the scope. However there are many cases when the scope has to be permanent and can be reentered later. This further complicates the matter.

The Lisp code generation phase produces Common Lisp, which is compiled using the lisp compiler in the Lisp Machine. The compiler compiles directly into Lisp Machine virtual memory. The compiled routines can then be called using the *execution commands.*

The Intel code generation phase produces EM code, which has to be moved in to a Unix machine having the ACK installed. It is further compiled by ACK Intel code generator to a Intel assembler file, which can be compiled with Intel assembler to a 8080 object code.

3 Division into packages and modules

A list of all source modules follows. The modules are grouped to packages according to their interrelationships. Each module corresponds to one source file and packages usually correspond to directories. The exact location of the source code in lisp-machine file system is listed in file *sys:site;newxe.translation.*

The **utility package** is implemented by the following modules: *dump, compatibility, util, name, hash, point, find-module*

The **abstract syntax** package implements the tools for specification and manipulation of abstract syntax trees. It consists of modules *ast, past, cast, past-equal, cast-equal, ppast*, and, *pcast.*

The **error handler** package implements the error handling facility. It consists of the module *error-handler.*

The **parse package** implements the scanner and the parser of the XE-language. It consists of modules *parser-macros, reserved, parser-funcs, scanner,* and *parser.*

The desugaring of the XE syntax is done by the **desugaring** package which consists of the module *desugar.*

The instantiation of generators is done by copying the PAST syntax tree and then substituting the parameter by parameter value. The substitution is done by **substitution** package which consists of the module *past-subst.*

The **descriptor package** implements the descriptor generator and, using it, defines the central data structure of the compiler. It consists of modules *desc-gen, desc-method, lazy, desc, init-desc, desc-aux, phase, print-desc, describe-desc, examine-desc,* and *find-routine.*

The checker is the central part of the XE compiler. It checks the semantic validity of XE programs and transforms the PAST structures to CAST structures. The **checker** package consists of modules *status, type, refs, checker-aux, checker-aux-funcs, checker-expr-aux, check-primitive, check-expression, check-instance, check-statement, assign-check, check-module, instantiate,* and *checker-top-level.*

The **context** package implements the context mechanism of the XE compiler. It consists of modules *generation, symbol-table, context,* and *scope-print.*

The **database** package implements the database of software objects. It consists of modules *module-base, generation-base,* and *module-base-browser.*

The environment has a smart recompilation algoritm. It is implemented by the **recompilation** package, which consists of modules *pdiff, affected, and scope-copy.*

The XE compiler has two code generators. The lisp code generator is implemented by the **lisp code generator** package, which consists of modules *libcall, libdef, cg-util,* and *codegen.* The intel code generator is implemented by the **intel code generator** package, which consists of modules *icg-util, icg-modules, icg-patterns, icg-optimize,* and *icg-main.*

The ZMACS editor is the main interactive component in the environment. It is customized for the XE language by the **zmacs** package, which consists of modules *xe-mode* and *zmacs-init.*

The XE compiler runs in a process of its own. The process is implemented by the **process** package, which consists of modules *process-aux* and *process.*

The top level user interface joining together the editor, compiler and error handling facility is implemented in the **user interface** package. It consists of the module *frame.*

Mosts of the datatypes used in the run time environment for the XE language in the lisp machine map directly to lisp data types. The auxliary facilities needed to implement the stream datatype are implemented by the **lisp-library** package, which consists of the module *_stream.*

4 Utility Package

The **utility package** is implemented by the following modules: *dump, compatibility, util, name, hash, point, find-module, main, profile-aux, freq, time, stack, queue.* It provides auxliary facilities used by the other parts of the environment.

4.1 Dump

The dump module is used to backup any system to another machine. This is done by calling the following function:

Function: `dump-system-to-unix`
Arguments:

```
  system &key newxe
          unix-target-directory "hila:/t/users/enuutila/newxe"
          action #'zl:copyf
          copy-always-p nil
          dump-time-file-name "touho:>newxe>src>dump-times.lisp"
          additional-files nil
          verbose-p nil
          info-stream t
          lspp T
          version :newest
```

Documentation: `System` is the system to be dumped, `action` is the function to be applied to each file, `copy-always-p` determines whether to copy even when the file is unchanged between this copy and previous one, `dump-time-file-name` is the name of the file where the dump time is written, `additional-files` is a list containing the additional files to be dumped in addition to the files of the system, the `verbose-p` determines whether to write any information during the operation, the `info-stream` is the stream to write the information to, the `lspp` determines whether to change the file name endings from “lisp” to “lsp”, and `version` is the version of the system to be dumped.

This module also contains a few shorthand functions for the most common backups.

4.2 Util

The utility module contains a set of extensions to Common Lisp.

Macro: `compiler-error`
Arguments:

```
     &rest args
```

Documentation: `Args` is the argument list for an error when `debugxe` is set, otherwise the macro expands to nil.

Macro: `test-compiler-error`
Arguments:

    cond
    &rest args

Documentation: If `debugxe` and `cond` are true signals an error using arguments `args` otherwise returns nil.

Macro: `push-list`
Arguments:

    list
    place

Documentation: Appends the list `list` to the list `place` and setfs the result to be the new value of `place`.

Function: `circular-list`
Arguments:

    &rest elements

Documentation: Returns a circular list containing the elements `elements`.

Macro: `while`
Arguments:

    s
    &body body

Documentation: While test `s` returns a true value executes `body`.

Macro: `downdotimes`
Arguments:

    var
    times
    &optional return
    &body body

Documentation: Executes `body` `times` times. `var` will have values of a decreasing sequence from `(- times 1)` to `0`. If `return` is given, returns its value at the end of the sequence.

Function: `before-if`
Arguments:

    fn
    list
    &key key #'identity

Documentation: Returns the part of the list `list` preceding the first element, which extracted with `key` returns true when `fn` is applied to it.

Function: `every-second`

Arguments:

`list`

Documentation: Returns a list containing every second of the elements of the list `list`.

Function: `concs`
Arguments:

`&rest strings`

Documentation: Returns a string containing the strings `strings` concatenated.

Function: `only`
Arguments:

`list`

Documentation: Checks that the length of list `list` is one and returns its element.

Function: `new-list`
Arguments:

`size`
`fn`

Documentation: Creates a new list of length `size` containing the elements returned by function `fn`

Function: `power`
Arguments:

`x`
`n`

Documentation: Returns the value of x^n.

Function: `transitive-closure`
Arguments:

```
initial-set
childsfn
&key test #'eql<
     reflexivep T
```

Documentation: Returns the set of the elements reached from the `initial-set` using function `childsfn`. Uses `test` to remove equal elements from the resulting set. If `reflexivep` is true returns also the initial elements.

Macro: `dolists`
Arguments:

`decls`
`&body forms`

Documentation: A dolist which takes multiple variable/list pairs as `decls`, binds the variables to the elements of the lists and executes the `forms` as long as none of the lists has ended.

Function: `gunion`

Arguments:

`&rest args`

Documentation: Returns a list, which is the union of the elements of the lists given as `args`.

Macro: `output-to-string`
Arguments:

`&body forms`

Documentation: Executes the forms `forms` and returns a string which contains all the output, which has been output to the stream `*standard-output*`.

Function: `to-keyword`
Arguments:

`sym`

Documentation: Interns the symbol `sym` into the keyword package.

Function: `intern-format`
Arguments:

`format`

`&rest args`

Documentation: Interns the string created by format giving `format` as the format control string and `args` as arguments.

4.3 Name

This module implements the datatype `name`, which is used throughout the compiler. Name is a pair of an uppercase string and a hash key. It is used to diminish the amout of hash key calculations.

The operations are:

name$create	string, (hash-key) $\longrightarrow$ name
name$string	name $\longrightarrow$ string
name$hash	name $\longrightarrow$ hash-key
name$equal	name, name $\longrightarrow$ boolean
name$less	name, name $\longrightarrow$ boolean
name$concat	name,... $\longrightarrow$ name
name$name-p	anything $\longrightarrow$ boolean

4.4 Hash

The hash module implements the hash table datatype. It is a straight forward hash table using a list of buckets for elements with the same hash key. The hash table is expanded automatically as the rehash point is reached. The size of the hash table is always a prime number. The table is highly parameterized. Its representation is a struct `xe-hash-table`, which has fields

elements	: an array containing the data
size	: the size of the array
element-count	: the number of data elements it the table
rehash-point	: the number of elements when the next rehash is to be made
rehash-ratio	: the fill-in ratio when the next rehash is to be made
key-datum-=	: the function to use to test for key equality
hash-key-function	: the function to be used to compute a hash key
chain-assoc-function	: the function to be used to assoc an element from a bucket list

The operations are:

Function: `hash-table$create`
Arguments:
```
initial-size
&keyrehash-ratio 0.8
    key-datum-= #'string=
    hash-key-function #'string-hash-key
    chain-assoc-function #'string-assoc
```
Documentation: This function returns an empty hash table. The `initial-size` is the low limit for the initial size of the table. The `rehash-ratio` is the fill in ratio point for rehash. The `hash-key-function` is a function from the key type to integer computing the hash key, and the `chain-assoc-function` is a function from a key type, alist pair to an element.

Function: `hash-table$rehash`
Arguments:
```
table
```
Documentation: Rehashes the `table` given as argument. Returns nil.

Function: `hash-table$find`
Arguments:
```
table
key-datum
&optional hash-key
```
Documentation: Returns the element from the hash table `table` corresponding to the key `key-datum` using `hash-key` as hash-key.

Function: `hash-table$remove`

Arguments:
  `table`
  `key-datum`
  `&optional hash-key`
Documentation: Removes the element from the hash table `table` corresponding to the key `key-datum` using `hash-key` as hash-key. Returns nil.

Function: `hash-table$map`
Arguments:
  `function`
  `table`
Documentation: Maps the function `function` to the elements of the array `hash-table$elements`.

4.5 Point

The datatype point is used to represent locations in XE source code. It is implemented as a list of `(line-number index filename-cell)` tuples. The filename-cell is a cons-cell. For each file, there is only one filename-cell, which makes it possible to change the name of the file without going through all points.

The operations for the **filename-cell** are:

filename-cell$create	filename ⟶ filename-cell
filename-cell$filename	filename-cell ⟶ filename

The operations for **point** are:

point$create	integer, integer, filename-cell ⟶ point
point$line-number	point ⟶ integer
point$index	point ⟶ integer
point$filename-cell	point ⟶ filename-cell
point$filename	point ⟶ filename
point$=	point, point ⟶ boolean
point$<	point, point ⟶ boolean
point-p	anything ⟶ boolean

4.6 Find

The module find contains routines used mainly for system development purposes. They allow one to inspect the state of the `top-level-name-scope`. There are tree types of names: string, string!string, and string$string. The exclamation mark is used to separate the datatype and its routine when the routine is not visible outside and the dollar sign when the routine is visible outside.

The function to call is:

cp$send-find-module	string ⟶ value, constant

4.7 Queue

The module `queue` contains two datatypes. A priority queue named `queue` and a usual queue named `cqueue`.

The datatype queue is implemented as a struct with fields

name	the name of the queue, a string
compare	the comparison function
contents	the contents as a list

The operations are:

queue$insert	queue, element-type ⟶ element-list
queue$insert-to-tail	queue, element-type ⟶ element-list
queue$priority-insert	queue, element-type ⟶ element-list
queue$first	queue ⟶ element-type
queue$remove-first	queue ⟶ element-type
queue$remove-element	queue, element-type ⟶ element-list
queue$empty-p	queue ⟶ boolean
queue$member-p	elem-type, queue ⟶ boolean

The **cqueue** is implemented as a struct with fields

name	the name of the queue, a string
elements	the contents as a list

The operations for **cqueue** are:

cqueue$new	name ⟶ cqueue

cqueue$insert	cqueue element-type ⟶ element-type
cqueue$remove-element	cqueue element-type ⟶ element-list
cqueue$remove-first	cqueue ⟶ element-type
cqueue$empty-p	cqueue ⟶ boolean
cqueue$first	cqueue ⟶ element-type

4.8 Compatibility

The compatibility module aims to remove the differences between lisp systems. It interns a few symbols to different packages according to the lisp-machine, defines a function, unlocks the lisp package etc. The code is straightforward.

5 The Abstract Syntax Package

The ast package contains the module implementing the AST language, and the modules past, past-aux, past-equal, ppast, cast, cast-equal, pcast, and split, which use the AST language.

5.1 The AST Language

AST is an object oriented language for the definition of abstract syntaxes. Abstract syntax tree nodes are objects, and methods can be applied on these objects. The structure of the abstract syntax defines a simple three level inheritance hierarchy. The language is documented in [4]. A throughout reading of [4] is strongly recommended.

An abstract syntax consists of a set of *productions*. A production defines a *domain* and a set of possible *options* of that domain. The options consist of a name and a sequence of *fields*. Every field has a name and a type. The type may be a domain, meaning that the field must have a value and the value must be a node of the given domain, or `&opt` domain, meaning that if the value is given it must be a node of the given domain, or `&list` domain, meaning that the value must be a list of nodes of the given domain, or `&list1` domain, meaning that the value must be a non-empty list of nodes of the given domain. There may also be *auxiliary fields* that may contain data that is not of any abstract syntax domain, e.g., attribute values.

The AST compiler compiles the abstract syntax definition into a set of Common Lisp functions. For every option of a production a creator function with name *syntax$*`create`-*domain-option* is created. For every field of an option an accessor function with name *syntax$domain-option-field* is created. Also a set of type predicates is created. For every domain a type predicate with name *syntax$domain*-`p` is created, and for every option of a domain a type predicate *syntax$domain-option*-`p` is created. The accessor and creator functions check that the types of their arguments are legal.

It is also possible to define generic *methods*. A *generic function* is defined and then for every node type a method is written that implements the generic function. The generic function is called whenever we must branch according to the type of the node. When a new node type is defined one only has to write methods for that node type. Thus, the effects of defining (or deleting or changing) a node type are local.

There is also a *wrapper* facility that can be used for the redefinition of a method. A wrapper is a piece of code that surrounds the actual method code. The wrapper code may, for example, examine the values of the arguments of the method, modify some of them, and then call the actual method.

The *structural methods* allow the programmer to effectively write routines that manipulate entire syntax trees. A structural method is a macro that expands to a set of methods, one for each node type. In a structural method definition the programmer defines abstractly how nodes should be processed according to their structure.

5.2 The Operation of the AST Program

The abstract syntax is defined using macro **define-abstract-syntax**. The form of the macro is:

Macro: `define-abstract-syntax`
Arguments:

```
syntax-name
&key productions
     start-domains
     check-types-p T
```

Documentation: `syntax-name` is the name of the syntax to be defined (a symbol), `productions` is a list of productions (described later), `start-domains` is a list of start domains, and `check-types-p` determines whether the code for type checks is to generated or not.

The syntax of the productions is:

```
productions ::== '(' {production} [global-aux-defs] ')'

production ::== '(' domain option {option} [domain-aux-defs] ')'

option     ::== '(' {argument} {option-aux-def} ')'
            |   '(' ':'option-name {argument} [opt-aux-defs] ')'

argument   ::== arg-name
            |   '(' arg-name type ')'
            |   '(' arg-name domain ')'
            |   '(' arg-name type domain ')'

type       ::== '&opt'
            |   '&list'
            |   '&list1'

global-aux-defs ::== aux-defs

domain-aux-defs ::== aux-defs
```

```
opt-aux-defs ::== aux-defs

aux-defs   ::== '&aux'
    '(' aux-name aux-type ['&test' test-function-name] ')'
     { '(' aux-name aux-type ['&test' test-function-name] ')' }

test-function-name ::== symbol (unquoted)
```

The expansion of the definition creates a set of definitions:

The routines to **create** the nodes of the tree:

```
<syntax>$create-<domain>-<option-name>
    {:<global-aux-def>}
    {:<domain-aux-def>}
    {:<option-aux-def>}
    {:<arg-name> }
    -->
    object of type <syntax>$<domain>-<option-name>
```

The routines for **reference**. For each one of them a **setf method** is generated.

```
<syntax>$<domain>-<option-name>-<arg-name>
    object of type <syntax>$<domain>-<option-name>
    -->
    the argument

<syntax>$<global-aux-def-name>
    object
    -->
    global-aux-def

<syntax>$<domain>-<domain-aux-def-name>
    object
    -->
    domain-aux-def

<syntax>$<domain>-<option-name>-<option-aux-def-name>
    object
    -->
    option-aux-def
```

The generic functions for **syntax, domain, and type queries**.

```
object-syntax
    object
    -->
    <syntax>

object-domain
    object
    -->
    <syntax>$<domain>

object-type
    object
    -->
    <syntax>$<domain>-<option-name>
```

The **type check predicates**:

```
    <syntax>$<domain>-<option-name>-p
    object
    -->
    t | nil
```

The nodes of the abstract syntax tree are represented as arrays 3.

Figure 3: The object structure

and the following properties are set for the symbols corresponding to the names in syntax.

```
<syntax>
    :ast        ast%ast
    :syntax     <syntax>
    :ast-type   :syntax

<syntax>$<domain>
    :syntax     <syntax>
    :ast-type   :domain

<syntax>$<domain>-<option>
    :syntax     <syntax>
    :ast-type   :option
```

The `syntax` entries refer to the datastructure:

```
ast%ast =      record (:conc-name ast%)
                 name: symbol
                 check-types: bool
                 start-domains: list of domain-name
                 productions: list of ast%production
                 aux-defs: list of ast%aux-def
               end

ast%production =
               record (:conc-name ast%prod-)
                 domain: domain-name
                 options: list of ast%option
                 aux-defs: list of ast%aux-def
               end

ast%option =   record (:conc-name ast%opt-)
                 name: symbol | nil
                 args: list of ast%argument
                 aux-defs: list of ast%aux-def
               end

ast%argument =
               record (:conc-name ast%arg-)
                 name: symbol
                 domain: domain-name
                 kind: arg-type
```

```
                  accessor
                end

arg-type =      '&opt | '&list | '&list1 | '&plain

ast%aux-def = record (:conc-name ast%aux-)
                  name: symbol
                  type: xe-type
                  test: function-name | nil
                  accessor
                end

domain-name = symbol
```

The generic methods are defined by:

```
(defmacro define-abstract-syntax-method
        ((method-name object-name) arglist &body body)
```

which defines a function:

```
(defun <method-name> (self <arglist>)
   body)
```

for all the objects below `<object-name>`, i.e., if the corresponding object is of type `:syntax` the function is defined for all objects, if it is of type `:domain` function is defined for all options of that domain, and if it is of type :option it is defined only for that option. Functions are ordered so that the most specific one overrides others.

The **wrappers** are defined using the macro:

```
(defmacro define-abstract-syntax-wrapper
 (method-name node-type inner-function-variable)
args &body body)
```

It redefines the corresponding method to be the one given by `body`. The previous definition is callable from the new method using name `inner-function-variable`.

The **structural** methods are defined using the macro:

```
(defmacro define-abstract-syntax-structural-method
  ((method-name syntax &optional option-names) arglist &body body))
```

It defines a method for each node of the syntax `syntax`. The method contains the code of `body`. During the evaluation of `body` the variables `syntax, domain, option, option-fullname, option-creator, args, option-aux-defs, domain-aux-defs, global-aux-defs, and start-domain-p` are bound.

5.3 PAST and CAST

The abstract syntax trees PAST and CAST are defined in modules *past* and *cast* respectively. The routines printing them are defined in modules *print-past* and *print-cast*. They define *ppast* and *pcast* methods for the nodes of the syntax trees. The implementation is done by writing a method for each node type. The modules *past-equal* and *cast-equal* define structural methods for testing the equality of the syntax trees.

6 The Error Handler Package

The error handler package contains module *eh*. It defines the format of error messages, the structure of error messages, and the routines to handle them.

The error messages are represented as structures:

```
(defstruct error-msg
  msg
  point
  generator-point)
```

where `msg` is the text of the error message, `point` is the erroneous point in the source code, and `generator-point` is the point in the respective generator, if the error was found in an instance.

The errors in modules are represented using structure:

```
(defstruct (module-errors (:conc-name me$)
  generation
  module-name
  msgs)
```

where `generation` is the generation of the module, `module-name` is the name of the module, and `msgs` contains the list of error-messages.

Errors are collected also into respective constant descriptors. The constant being checked is bound into `*error%current-constant*` and the erroneous modules into `*error%errorneous-modules*`.

Error messages are inserted into the system by function:

Function: `error-message`
Arguments:

```
context
format-string
point
&rest format-args
```

Documentation: `context` is the context of the erroneous location (can be nil), `format-string` is value of an error-constant, `point` is the erroneous location, and `format-args` are the arguments for the `format-string`.

The error messages associated with a constant can be printed using:

Function: `print-error-messages-of-constant`

Arguments:

```
constant
&optional stream *standard-output*
```

Documentation: Prints the error messages associated with `constant` to the `stream`.

An erroneous module can be found from the list of erroneous modules using the function:

Function: `me$find`

Arguments:

```
generation
module-name
&optional createp nil
```

Documentation: Searches for a module called `module-name` in generation `generation` from `*error%errorneous-modules*`. If `createp` is set, creates a module-error-struct for the corresponding module with an empty error-msgs list.

7 The Parser Package

The **parser package** is implemented by following modules: *scanner, parser, parser-funcs, parser-macros, reserved.* It implements the lexical and syntactical analyst for the XE language.

7.1 Scanner

`*pa%reserved-words*`

This variable holds the hash table for the reserved words. The hash table is constructed so that no conflicts arise.

`pa%read-char-by-lines`
`pa%unread-char-by-lines`

These functions read and unread characters. The reading is implemented using the Lisp function *read-line* for efficiency. Variables are updated to indicate the current position in the input stream.

`pa%read-number`
`pa%read-symbol`
`pa%read-string`
`pa%read-character`

These functions read XE numbers, identifiers, strings and characters.

`pa%lexical-read`

This is the lowest level of the scanner. It reads characters and recognizes tokens according to the first character.

`pa%lexical-token`

This function recognizes those tokens that consist of two characters, e.g. `<=`.

`pa%lexical`

This is the top level of the scanner. It recognizes also keywords.

7.2 Parser

This is the syntactic analyst for the XE language. The analyst is in the function **pa%xe-parser**. It is implemented using ROCC [8].

7.3 Parser-funcs

This module contains functions for handling errors in the input.

```
pa%correct-tpos
pa%seen-error
pa%check-last-error
pa%give-fatal-error-message
pa%give-nonfatal-error-message
pa%corrected-error
```

These functions handle the error messages for fatal and corrected errors.

```
pa%definition-error
pa%statement-error
pa%fatal-statement-token-p
pa%fatal-definition-token-p
```

These functions handle the skipping of input when a fatal error has occurred. For definitions, the input is skipped to the next definition (`id = proc/iter/datatype`). For statements, the input is skipped to the next *end* word.

```
pa%show-token
```

This function is used for printing a token description to the user.

```
pa%lexical-value-of-token
pa%delete-cost
pa%insert-cost
pa%exchange-cost
```

These functions are the cost functions used in the error correction algorithm of ROCC.

```
pa%split-error
pa%split-statement-seq
```

These functions are used in splitting constant and statement sequences into two parts.

```
pa%past-has-error-p
```

This structural PAST function traverses a PAST structure and returns T if there is an error node in the structure. Error nodes are present if there has been a fatal error in the parsing.

```
pa$parse-file
pa$parse-string
pa$parse-stream
```

These functions parse input from files, strings and streams. They return the PAST structure.

7.4 Parser-macros

Declares a number of variables for holding the state of the parser and the scanner.

```
pa%create-token-position
pa%token-position-position
pa%token-position-data
pa%tpos
pa%tdat
pa%retpos
```

These macros handle the tokens and positions for the scanner.

```
pa%lexical-reading
```

This form binds the necessary variables before the scanner or parser can be used.

7.5 Reserved

The *reserved* module declares variables for the reserved words and tokens of the scanner and parser.

`*pa%reserved-words-alist*`

An alist of the reserved word identifiers for the parser and the reserved words as strings for the scanner.

`*pa%other-tokens-alist*`

An alist of the identifiers for other tokens for the parser and the strings describing these tokens.

`*pa%assocs*`

Declares the association and precedence rules for the arithmetic operators.

8 The Desugaring Package

The desugaring package contains routines, which transform a **tagcase**-statement to a **case**-statement and a **rule** to an **iterator**. First the rule transformation is explained:

```
condition => expr, ...

is comp[condition, yield(expr, ...)]
```

where

```
comp[condition-for, yield]
    for iteration do
        *comp[condition, yield]*
    end

comp[condition-clause, yield] ::=
    begin
       *comp[clause, exit #success, NONE]
    end
       except when #success: *yield* end

comp[clause-expr, success, failure] ::=
    if expr then *success* else *failure* end

comp[clause-all, success, failure] ::=
    begin
        for iteration do
            *comp[clause, NONE, exit #fail_exit]*
        end
        *success*
    end
        except when #fail_exit: *failure* end

comp[clause-some, success, failure] ::=
    begin
        for iteration do
            *comp[clause, exit #succ_exit, NONE]*
        end
        *failure*
    end
        except when #succ_exit: *success* end
```

```
comp[clause-not, success, failure] ::=
    *comp[clause, failure, success]*

comp[clause-and, success, failure] ::=
    *comp[clause1, comp[clause2, success, failure], failure]*

comp[clause-or, success, failure] ::=
    *comp[clause1, success, comp[clause2, success, failure]]*

comp[clause-parenthesis, success, failure] ::=
    *comp[clause, success, failure]*
```

A rule is transformed by function **ds$rule** and a tagcase statement by function **ds$tagcase**.

9 Instantiation Package

Routine and data type generators are *instantiated* to create modules that can be used like normal non-parameterized modules. A generator is provided with actual constant values for the formal parameters declared in its header. At instantiation, the occurrences of the formal parameters in a module are substituted by the values of the actual parameters, and the parameter declarations and the possible **where** clauses are deleted.

In the XE programming environment the substitution of formal parameters is performed at the PAST level. Thus, when a generator is instantiated the PAST representation of the instance is build from the PAST representation of the generator. The information stored in the descriptors of the generator is not used to determine the structure of the instance. The resulting PAST is processed by the checker as a normal non-parameterized module to form the descriptors and CAST representation needed. An implementation that would directly use the descriptors of the generator, although it would be faster, was considered too complex to write.

There are several different kinds of instantiations. A generator can have a list of ordinary formal parameters (i.e., it is an *ordinary generator*), or it can have a formal field list as its parameter (i.e., it is a *selector generator*). Data types can *inherit* generators. An inherited generator needs special treatment.

The actual parameter values can all be simple constant values (constructed from literals), or some of them can be formal parameters of an other module. In the former case the instance is called *normal*, in the latter case it is called a *pseudo-instance*. Typically the representation type of a data type generator is a pseudo-instance of some other data type generator. Sometimes a data type generator refers to itself with the same parameter configuration as in its header (i.e., it refers to its *own pseudo-instance*) In these cases no instance is made: the generator itself can be used as its own pseudo-instance.

The **instantiation** package consists of modules *instantiate* and *past-subst*. The instantiate module deduces what kind of an instance is to be made and collects the formal/actual parameter pairs. Past-subst performs the construction of the PAST of an instance. Mostly the construction of an instance PAST is performed by copying the generator PAST node by node. Some special nodes are deleted (e.g., declarations of formal parameters) and some substituted by others (e.g., occurrences of formal parameters).

9.1 The instantiator

The instantiation is performed by

Function: `check%instantiate`
Arguments:

    generator
    param-values
    point

Documentation: Returns the descriptor of the *param-values* instance of *generator*. It should be noticed that although the type of the instance returned is usually a `desc$instance` it can also be a `desc$generator`. The *generator* must be a `desc$generator`. If the *generator* is an ordinary generator then *param-values* must be a list of `desc$param-values` and the number of them must match with number of parameters of the *generator*. If the *generator* is a selector generator then the *param-values* must be a `desc$field-list`. Legality of the parameter values should be checked before calling `check%instantiate`. *point* is the PAST point of the instantiation location (the place where the generator is used).

For both selector generators and ordinary generators the function `check%instantiate` first checks if the instance is an own pseudo-instance or if it already exists. If not, it creates a new descriptor for the instance, generates an internal name for the instance, calls `subst$past` to create a new PAST for the instance, gives the result to `check%check-constant` (the checker), and returns the checked instance.

The instantiate module contains a number of other routines that are briefly described below.

Function: `subst$past`
Arguments:

    gen
    gen-past
    inst
    point

Documentation: Builds a new PAST from the generator *gen* whose PAST is *gen-past*. *Point* is the instantiation point and *inst* must be a new instance descriptor. If the generator is a selector generator, an internal field list constant for the parameter values is build.

Function: `check%collect-uses`
Arguments:

    param-values

Documentation: Collects **use** definitions given in the parameter values. Returns a list containing (*type* . *list-of-equates*) pairs, where *type* is a `desc$type` and *equates* are `desc$constants`.

Function: `check%equal-param-value`
Arguments:

    pv1
    pv2

Documentation: Returns `NIL` if the parameter values *pv1* and *pv2* are not equivalent (as defined in the XE Reference Manual).

Function: `desc$similar-field-list-p`
Arguments:

```
fl1
fl2
ordered-p
```

Documentation: Returns `NIL` if the parameter values *fl1* and *fl2* are not equivalent (as defined in the XE Reference Manual). When *ordered-p* is `NIL` the order of the fields is neglected.

Function: `check%substs`
Arguments:

```
gen
inst
point
field-list-const
```

Documentation: Collects the actual parameters and generates constant definitions for them. Names (or actually `past$idns`) of these constants form the replacements for the formal parameters. The names are generated automatically, and are artificial. `Check%substs` returns a substitution data struct for `subst%past`. *Gen* is the generator to be instantiated, *inst* a partially initialized instance, *point* the instantiation point, and *field-list-const* the actual parameter if *gen* is a selector generator (otherwise its value is ignored).

9.2 The Substitutor

An instance PAST is build from a generator PAST by:

Function: `subst%past`
Arguments:

```
past
sa
point
```

Documentation: *Past* is the generator PAST, *sa* a struct containing substitution data, and *point* a source point. The source point is the source code location where the instantiation is performed. The PAST nodes of the instance are given this as their source point. If the checker detects errors in the PAST produced (it should find none) the error messages are given at the instantiation location.

There are three different cases of substitution. There exists a form of the substitution data struct for each of them. *Normal-subst* is used when an ordinarily parameterized generator is instantiated or pseudo-instantiated. *Selector-instance-subst* is used when a selector generator is instantiated. *Selector-generic-subst* is used when a selector generator is pseudo-instantiated. In all of these three cases the substitution struct contains the following fields:

generator-name Name of the generator (a `name`).

instance-past Name of the instance (a `past$expr-idn`). This is an artificial name generated by the compiler.

subst-alist An association list that maps the formal parameter names (of type `name`) to the actual parameters (of type `past$idn`).

pseudo-p Suppresses copying the bodies of the routines, because code is never generated for pseudo-instances. This is `nil` if a normal instance is to be made.

A normal-subst struct also contains:

parms Names of formal parameters.

A selector-instance-subst struct also contains:

field-list Name of the formal field-list parameter (a `name`).

field-names The names of the actual fields (of type `past$idn`).

field-types The types of the actual fields (of type `past$idn`).

field-name-alist An association list that maps the formal field names (of type `name`) to actual field names (of type `past$idn`). The subst-alist is used to map the formal field types (`name`) to actual field types (`past$idn`).

A selector-generic-subst struct also contains:

field-list Name of the field list (a `name`).

field-pseudo-type Pseudo type for the pseudo field.

The basic copy algorithm of PASTs is implemented by an abstract syntax structural method. The method just copies a PAST tree in depth first order. For special nodes this basic structural method is surrounded by abstract syntax wrappers that prevent the direct copying of the node. They either make a substitution or ignore the node.

For some nodes (the `past$equate-or-own-equate`, `past$expr-idn`, and routine nodes) the common structural method cannot be used. Therefore a unique method is defined for each of these node types.

At some nodes the substitution association lists must be updated, because new names are declared in the PAST. At nodes of the types `past$equate` and `past$statement-tagcase-foreach` nodes a new field name/type pair can be declared. The formal name and type associated with the actual values must be added into the substitution lists before entering the value of the equate or body of the tagcase statement. At exit the formal name/type pair must be removed from the association list, because the same names can be used for other purposes elsewhere.

For a group of similar nodes (e.g., the routine nodes) the abstract syntax wrappers or methods are defined using a common macro. This makes the code a little harder to read, but a lot easier to maintain.

10 Descriptor Package

The descriptor package contains the implementation of the descriptor language called DG, the descriptor definitions defined in DG language, and some auxiliary modules.

10.1 DG language

DG is a general object class definition language that contains multiple inheritance and methods. It is defined in modules *desc-gen*, and *desc-method*.

10.1.1 Introduction

As opposed to most object oriented languages, a DG definition defines a set of classes, not just a single class. This permits an easy implementation of the DG compiler, because all classes can be compiled at the same time. However, compilations after modifications last longer.

Here is an example of a DG definition:

```
(define-descriptors windows
   :structures
   ((window
     :fields ((name string)
              (top int)
              (bottom int)
              (left int)
              (right int)
              (superior (&opt window))
              (inferiors (&list window))))
    (process
     :fields ((top-level-function (&union symbol lambda))))
    (bordered-window-mixin
     :instantiable nil
     :fields ((border-thickness int)))
    (bordered-window-process
     :parents (window process bordered-window-mixin))
    (:extern int :test fixnump)
    (:extern symbol :test symbolp)
    (:extern lambda :test (lambda (x)
                            (and (listp x)
                                 (eq (first x) 'lambda))))))
```

The name `windows` identifies the definition. It is used as a prefix in the creation of operation and class names. The list after keyword `:structures` contains the class definitions. In the example seven classes are defined, namely `window`, `process`, `bordered-window-mixin`, `bordered-window-process`, `int`, `symbol`, and `lambda`. A class definition consists of a set of field definitions and an optional parent class definition. A parent class definition contains one or more class names, thus multiple inheritance is permitted. A field consists of the name and the type of the field. The type is either a class or a type expression built of classes and type expression operations. Type expression operations are `&opt` meaning an optional field, `&list` meaning a list field, and `&union` meaning a disjoint union. For example, the value of the field `inferiors` of an object of the class `window` is a list of objects belonging to class `window` or to some of its descendant classes. Type expressions may be nested.

An object of a class has the fields that are defined in the definition of that class and all the fields that are defined in the ancestors of the class. Fields can not be redefined in subclasses.

Because DG is not a self contained language there must be a facility for using objects of other kinds as field values of DG objects. This is accomplished with the `:extern` class definitions. An extern class definition contains the name of the extern class and optionally a type predicate. Extern classes may be used in field type definitions in the same way as a normal DG class. However, an extern class may not be the parent of some other class. In our example the classes `int`, `symbol`, and `lambda` are extern.

Some classes represent behavior that is only reasonable when that class is combined with some other class. In our example, the class `bordered-window-mixin` is such a class. When we combine this class with some window class we can make windows that have border lines of variable width. It is not reasonable to have objects of bordered class and therefore the class definition contains the definition `:instantiable nil` which prohibits instantiation.

The DG compiler compiles the class definitions to a set of Common Lisp functions. For every class a creator function with name *dg-name*`$create-`*class-name* is constructed, where *dg-name* is the name that identifies the definitions. Also a type predicate with name *dg-name*`$`*class-name*`-p` is created. For every field name that occurs in the class definitions a field accessor function with name *dg-name*`$`*field-name* is created. The accessor functions are polymorphic so that the same functions can be applied to objects of different classes.

Methods can be written that manipulate DG objects. The DG methods resemble AST methods. First a generic function is defined and then for every class that should have the corresponding behavior a method is written that implements the behavior. When a method should be applied to an object the generic function is called, it finds out the correct method and calls it.

The generic function is defined with the macro `dg$define-method-function`. The form of the definition is

`(dg$define-method-function` *generic-function-name* `arg1 arg2` ...)

The methods are defined with macro `dg$define-method`. The form of the definition is

`(dg$define-method (`*dg-name method-name class name*`)` *args*
body`)`

As in AST methods the variable `self` is bound to the object that the method is applied to.

Like the AST language the DG language contains a structural method facility.

Access oriented functions are needed when the access of a field of an object should cause a side effect, for example the call of some trace function. Access oriented functions can also be used for the implementation of laziness. Because this is the main use for access oriented functions in the XE compiler we use the term *lazy field* when referring to fields that are associated with access oriented functions. Lazy fields can be defined in a class definition by listing the lazy fields accompanied with the names of functions that return a piece of Common Lisp code that implements the access oriented behavior.

Here is an example. Let the function `superior-accessor` be defined in the following way.

```
(defun superior-accessor (self direct index field-name)
  (let ((value-var (make-symbol "VALUE")))
    `(let ((,value-var (svref ,direct ,index)))
       (format t ``~%~A of ~A is ~A''
          ',field-name
          (window$name self)
          (window$name value))
       value)))
```

If we now replace the definition of class window of our class definition example with the following class definition

```
    (window
     :fields ((name string)
              (top int)
              (bottom int)
              (left int)
```

```
            (right int)
            (superior (&opt window))
            (inferiors (&list window)))
  :accessors (((superior) superior-accessor)))
```

then every time the superior field of an object of type `window` or some of its descendant classes is accessed a trace message will be printed.

10.1.2 Syntax of DG

Desriptors are defined using the **define-descriptors** macro:

```
(defmacro define-descriptors
    name
    (&key equates
          structures
          (all-initially-optional nil)
          (check-types-p T)
          (xe-file-name nil))
```

Where the syntax of `structures` is:

```
structures ::== '(' { structure } ')'

structure  ::== external | internal

internal   ::== '(' sname
                    [ ':instantiable' 't' | 'nil' ]
                    [ sypertypes ]
                    [ fields ]
                    attributes ')'

external   ::== '(' ':extern' sname [ :test sexpr ]
                 attributes ')'

sname      ::== symbol

supertypes ::== ':parents' '(' { sname } ')'

fields     ::== ':fields' '(' { field } ')'
```

```
attributes ::== { key-symbol sexpr }

field      ::== '(' fname ftype
                    [ ':settable' 't' | 'nil' ]
                    [ ':gettable' 't' | 'nil' ]
                    [ ':inittable' 't' | 'nil' ] ')'

fname      ::== symbol

ftype      ::== sname
            |   '(' '&list' ftype ')'
            |   '(' '&list1' ftype ')'
            |   '(' '&opt' ftype ')'
            |   '(' '&union' ftype { ftype } ')'
```

and the syntax of `equates` is:

```
equates    ::== '(' { equate } ')'

equate     ::== '(' name ftype ')'
```

10.1.3 Methods

A method calling function is defined with macro

```
(defmacro dg$define-method-function (fn &rest other-args)
```

It expands to a function, which funcalls the function corresponding to the `fn` in the descriptor given as argument.

Methods for single domains are defined using

```
(defmacro dg$define-method
        ((desc-name method desc-type) other-args &body body)
```

which defines a method called `method` for `desc-name desc-type`. The method takes as arguments self and `other-args` and has body `body`.

Wrappers are defined using

```
(defmacro dg$define-wrapper
          ((desc-name method desc-type previous-method-fn-var)
            other-args &body body)
```

Structural methods are defined using

```
(defmacro dg$define-wrapper
          ((desc-name method desc-type previous-method-fn-var)
            other-args &body body)
```

which defines one method for all desc types. The body of these methods is replaced by the body of this macro, which will be evaluated once for all desc types. During the evaluation of the body the following variables are bound to the descriptions of `desc-type`:

Variable	**Description**
parent	Parent descriptor.
name	Symbol for the descriptor type.
extern-p	T iff this method is for the external value.
creator	Creator function.
method	Symbol for the function about to be defined.
instantiable	T iff desc type is instantiable.
parents	list of parent descriptors, see below.
own-fields	list of field descriptors for fields defined in this particular descriptor type, see below for field descriptors.
all-fields	list of all field descriptors, even the ones inherited from parents.

Parent descriptors are not interesting as such, but they have operations that can be applied:

Function	**Description**
dg$parent-name	Symbol for the name of the parent.
dg$parent-extern-p	T iff parent is just an external field.
dg$parent-creator	Creator function for the parent. Note this is seldom used.
dg$parent-method	Function to be applied to the argument object of the structural processing if the structural method of the parent has to be invoked.

dg$parent-instantiable	T iff desc is instantiable.
dg$parent-parents	Parents of the parent.
dg$parent-own-fields	Own fields in the parent.
dg$parent-all-fields	All fields in the parent.

Field descriptors have operations:

Function	**Description**
dg$field-name	Symbol for the field name.
dg$field-kwdname	Keyword symbol for the field name. Can be used as an argument for the creator function.

dg$field-type	Type of the field:
	`<desc type name>`
	`(&extern <desc type name>)`
	`(&union <type list>)`
	`(&list <type>)`
	`(&list1 <type>)`
	`(&opt <type>)`

dg$field-method	Recursive calls to the structural method should use the value of this function as a function. See dg$parent-method.
dg$field-accessor	Returns the field access function for field f. This function may also be used with setf.
dg$field-sensitive-accessor	Just like dg$field-accessor, except that (get obj 'values)-type accessors are used.
dg$field-owner	Returns the desc type descriptor that defined this field. This name symbol of it is equal to name (bound variable) iff the field is defined in this desc type.

10.1.4 Implementation

`define-descriptors` uses `dg%parse-structure` to parse the definitions and to create the data structure described below to represent the syntactic structure of a definition. The property `:dg$desc` of the symbol named `desc-name` is then set to point to the `dg$desc`.

```
dg$desc (:conc-name dg$desc-)
  name          symbol (property :dg$desc -> .)
```

```
  equates        assoc-list of name . equate
  structures     list of dg$structure
  checkp         t | nil
  xe-file        not used

dg$structure (:conc-name dg$str-)
  name           symbol (property :dg$structure -> .)
  kind           :external | :internal
  definition-name
  instantiable   t | nil
  test           sexpr
  supertypes     list of symbol
  fields         list of dg$field
  attributes     list of dg$attribute
  field-index-alist
  accessors


dg$field (:conc-name dg$field-)
  definer
  name           symbol
  ftype          dg$ftype
  settable       t | nil
  gettable       t | nil
  inittable      t | nil

dg$ftype (:conc-name dg$ftype-)
  kind           :list | :list1 | :opt | :plain | :union
  subtypes       name | list of name

dg$attribute (:conc-name dg$attr-)
  key
  sexpr
```

The parser of the DG language consists of routines:

```
(defun dg%parse-ftype (ftype)
(defun dg%parse-structure (str)
(defun dg%parse-field (field)
(defun dg%parse-attribute (attribute)
```

The objects are represented as arrays 4:

The module also contains two printers. One to pretty print the data structure (called **dg$flat-print**) and one to print it in the original syntax (called **dg$print**).

Figure 4: The object structure

The functions which perform the operations associated with the descriptors are collected into lists:

```
*dg%misc-prelist*,  *dg%predicate-list*, *dg%create-list*,
*dg%accessor-list*, *dg%setf-list*, *dg%misc-postlist*
```

The type predicates are defined using function

```
(defun dg%define-type-predicate (str) ...)
```

the create function with

```
(defun dg%define-create-function (str) ...)
```

the accessors with

```
(defun dg%define-accessors () ...)
```

the setf methods with

```
(defun dg%define-setf-methods () ...)
```

The routines to access fields without trigging the lazy-field mechanism are:

```
(defun dg$get (obj field-name) ...)
(defun dg$set (obj field-name value) ...)
```

10.2 Desc definition

The descriptors are defined in module *desc* and the corresponding lazy accessors in module *lazy-accessor*. The phases used in compiler control are defined in module *phase*.

10.3 Other modules

The module *examine-desc*, *print-desc*, and *describe-desc* contain the examiner, printer, and describer.

11 The Checker Package

11.1 Checker Aux

The module **checker-aux** implements the checker method abstraction. The macros defined in this module hide the nasty details of error and scope handling so that they are not visible to the other parts of the checker.

The definitions are:

Form: `(define-abstract-syntax-generic-function check past)`
Documentation: Defines the main checker generic function `check`.

Form: `(define-abstract-syntax-generic-function check-header past)`
Form: `(define-abstract-syntax-generic-function check-body past)`
Documentation: Two generic functions that are used in the checking of such `past` nodes that must be checked in two phases, first the header and then the body, are defined.

Macro: `check%in-error-env`
Arguments:
`&body body`
Documentation: Creates an error handling environment. The environment has an internal error status flag that should have value non-NIL iff some errors have been encountered inside this error environment. Macros `check%error-test`, `check%error-test-exit`, `check%bool-test`, `check%bool-test-exit`, and `check%cast-test-exit` can be used for giving error messages and setting the error flag. The macro `check%return` should be used for returning a value from the environment. The `-exit` forms also exit from the error handling environment.

Macro: `defcheck`
Arguments:
`option`
`args`
`&body body`
Documentation: A macro that defines a `check` method. An error environment is wrapped around the `body` with `check%in-error-env`.

Macro: `define-header-check`
Arguments:
`option`
`args`
`&body body`
Documentation: A macro that defines a `check-header` method. An error environment is wrapped around the `body` with `check%in-error-env`.

Macro: `define-body-check`

Arguments:

    option
    args
    &body body

Documentation: A macro that defines a `check-body` method. An error environment is wrapped around the `body` with `check%in-error-env`.

Variable: `*check%error*`

Initial value:

Documentation: A variable that contains the unique error object that is used in the error handling.

Macro: `check%error-test`

Arguments:

    form
    &rest error-stuff

Documentation: If `form` evaluates to `*check%error*` an error message is given with
`error-stuff` as arguments and the error flag is set. Should be used lexically inside an error environment.

Macro: `check%error-test-exit`

Arguments:

    form
    &rest error-stuff

Documentation: If `form` evaluates to `*check%error*` an error message is given with
`error-stuff` as arguments and the environment is exited with `*check%error*` as value. Should be used lexically inside an error environment.

Macro: `check%bool-test`

Arguments:

    form
    &rest error-stuff

Documentation: If `form` evaluates to `NIL` an error message is given with `error-stuff` as arguments and the error flag is set. Should be used lexically inside an error environment.

Macro: `check%bool-test-exit`

Arguments:

    form
    &rest error-stuff

Documentation: If `form` evaluates to `NIL` an error message is given with
`error-stuff` as arguments and the environment is exited with `*check%error*` as value. Should be used lexically inside an error environment.

Macro: `check%cast-test-exit`

Arguments:

`form`

`&rest error-stuff`

Documentation: If `form` evaluates to something that is not a `cast` an error message is given with
`error-stuff` as arguments and the environment is exited with `*check%error*` as value. Should be used lexically inside an error environment.

Macro: `check%errors-p`
Arguments:

Documentation: Returns the value of the error flag. Should be used lexically inside an error environment.

Macro: `check%return`
Arguments:

`value-form`

Documentation: Exits from error environment. If the error flag is set the value is `*check%error*` otherwise the `value-form` is evaluated and the values are returned. Should be used lexically inside an error environment.

Macro: `check%return-ignore-errors`
Arguments:

`value-form`

Documentation: Exits from the error environment. The values of evaluating the `value-form` are returned. Should be used lexically inside an error environment.

Macro: `check%opt-check`
Arguments:

`past`

`&rest args`

Documentation: If `past` evaluates to non-NIL, method `check` is applied on it with `args` as other arguments.

Macro: `check%if-required`
Arguments:

`required-type`

`&body body`

Documentation: Evaluates body if `required-type` evaluates to a `desc` object. Used in the expression checker.

Macro: `check%if-peek`
Arguments:

`required-type`

`&body body`

Documentation: Evaluates body if `required-type` evaluates to `:peek-type`. Used in the expression checker.

Macro: `check%if-deduce`
Arguments:
```
required-type
&body body
```
Documentation: Evaluates body if `required-type` evaluates to `:deduce-type`. Used in the expression checker.

Macro: `check%in-scope`
Arguments:
```
(context
 equates
 scope-ref
 &optional (unordered nil)
           (redefine-p nil)
 &rest initial-things-to-insert)
&body forms
```
Documentation: A macro that enters a scope, evaluates `forms` and then exits the scope. First the `scope-ref` is evaluated and if it is `NIL` a new scope is created and `setf`ed to `scope-ref`. `unordered` and `redefine-p` are used as arguments for `context$open-scope`. The current name scope of `context` will be the newly created scope. The objects in `initial-things-to-insert` are inserted into the scope and then constant descs are created for every equate in `equates`. The constant descs are also inserted into the scope. If the `scope-ref` evaluates to a scope, it will be used as the current name scope of the `context` and unnecessary stuff is removed from the scope (see the documentation of `check%clear-scope`). After this the `forms` are evaluated, the scope is closed and the current name scope of the `context` is reset to its previous value.

Macro: `check%in-scope-with-iterate`
Arguments:
```
(context
 equates
 scope-ref
 &optional (unordered nil)
           (redefine-p nil))
&body forms
```
Documentation: Same as `check%in-scope`, but also an `:iteration` scope is entered before beginning and exited after the evaluation of `forms`.

Macro: `check%header-in-scope`
Arguments:
```
(context
 equates
```

```
  scope-ref
  &optional (unordered nil)
            (redefine-p nil))
 &body forms
```
Documentation: Same as `check%in-scope`, but the scope is not closed after the evaluation of the `forms` except when it is a *persistent* scope that can be safely closed without destroying its contents. Should be used in the checking of the headers of such PAST nodes that are checked in two phases, first the header and then the body.

Macro: `check%body-in-scope`
Arguments:
```
 (context
  scope-ref)
 &body forms
```
Documentation: The counterpart of `check%header-in-scope` for the checking of bodies of such past nodes that are checked in two phases, first the header and then the body. The scope is re-entered, the `forms` are evaluated and then the scope is exited and closed.

Struct: `check%phase`
Documentation: A `struct` type that is used in `check%module` for describing the phases of the checking.

Macro: `check%module`
Arguments:
```
 (required-phase
  constant-ref)
 &body forms
```
Documentation: This is a complicated control macro for the checking of the module PAST nodes, i.e. `past$proc`, `past$iter`, `past$rule`, `past$foreign-proc`, and `past$type-def`. The checking of the module past nodes is performed in ordered *phases* (they are defined in module `phase`). The forms in a phase compute some fields of the descriptors of the module. When a module descriptor is created the fields that the checker must compute are initialized with the phase names so that the lazy field facility (see the documentation of descriptors) can determine which phase has to be executed in order to get the actual value of the field. The `forms` are either phase names (keyword symbols) or Lisp forms. The last form should be a phase name. A phase consists of the forms between two phase names (or between the beginning of forms and the first phase name). The phase name after the phase forms names the phase. The macro splits `forms` into phases and builds a control structure for the evaluation of the phases. That control structure evaluates `required-phase` and `constant-ref` and then it evaluates all phases between the current phase of the constant and the required phase. Before the evaluation of the phases the constant is marked "not idle" in order to catch illegal recursion. After the evaluation of the phases the constant is marked "idle" and if the error flag of the error environment is set the status (see the documentation of module `status`) is set erroneous. Also the

current phase of the constant is updated. Should be used lexically inside an error environment.

Macro: `check%enter-scope`
Arguments:

```
(context
 equates
 scope-ref
 &optional (unordered-p nil)
           (redefine-p t))
```

Documentation: Inside a `check%module` macro call `check%in-scope` cannot be used, because the body may be entered and exited multiple times. Macro `check%enter-scope` can be used for entering scopes and macro `check%exit-scope` for exiting scopes in these cases.

Macro: `check%exit-scope`
Arguments:

```
(context
 scope-ref)
```

Documentation: Inside a `check%module` macro call `check%in-scope` cannot be used, because the body may be entered and exited multiple times. Macro `check%enter-scope` can be used for entering scopes and macro `check%exit-scope` for exiting scopes in these cases.

Macro: `check%setf`
Arguments:

```
&rest items
```

Documentation: If the error flag is not set behaves like `setf`, but if the flag is set `*check%error*` is used as the assigned value. Should be used lexically inside an error environment.

11.2 Checker Aux Funcs

The module **checker-aux-funcs** contains some auxiliary functions that are used in the checker.

Function: `check%check-list`
Arguments:

```
list
&rest args
```

Documentation: Maps `check` method to the elements of the list and returns the results as a list. If any of the results is `*check%error*` the error flag is set and the result of the function call will be `*check%error*`. Should be used lexically inside an error environment.

Function: `check%check-list-nconc`

Arguments:

```
list
&rest args
```

Documentation: Same as `check%check-list`, except that the results are expected to be lists and they are `nconced`. Should be used lexically inside an error environment.

Function: `check%error`
Arguments:

Documentation: Returns `*check%error*`.

Function: `check%error-p`
Arguments:

```
x
```

Documentation: Returns T if the argument is `*check%error*` and NIL if it is not.

Function: `check%create-constant`
Arguments:

```
equate
context
&key (queue-insert-p t)
     (scope-splice-allowed-p nil)
     (generation *system$current-generation*)
```

Documentation: Creates a constant corresponding to `equate` and inserts it into `context`. If `queue-insert-p` is non-NIL then the constant is also inserted into the checker queue. `scope-splice-allowed-p` is an argument for `context$insert` and `generation` an argument for `check%initialize-constant`.

Function: `check%create-undefined-constant`
Arguments:

```
name
context
```

Documentation: Creates an unitilialized constant called `name` and inserts it into `context`.

Function: `check%initialize-constant`
Arguments:

```
constant
equate
undefinedp
containing-context
generation
```

Documentation: Initializes the various fields of a constant.

Function: `check%constant-past`

Arguments:

    expr-past

Documentation:

Function: `check%initial-phase`
Arguments:

    equate

Documentation: Returns the initial phase of the checking (see the dcocumentation of `check%module`) corresponding to the given equate.

Function: `check%final-phase`
Arguments:

    equate

Documentation: Returns the final phase of the checking (see the dcocumentation of `check%module`) corresponding to the given equate.

Function: `check%delete-constant`
Arguments:

    name
    context
    &optional (error-if-not-found-p t)

Documentation: Unlinks and removes a constant from a context.

Function: `check%clear-scope`
Arguments:

    context
    ignore

Documentation: Removes all `descs` that are not constants from the name scope of `context`.

Function: `bag$create`
Arguments:

Documentation: Creates a bag.

Function: `bag$insert`
Arguments:

    bag
    element

Documentation: Inserts `element` into `bag`.

Macro: `bag$do`
Arguments:

    (elem
     bag)
    &body body

Documentation: Maps over elements of a bag. For every element of `bag` variable `elem` is bound to the element and `body` is executed.

Function: `bag$member`
Arguments:
    `item`
    `bag`
    `test`
Documentation: Tests whether `item` is a member of `bag` using `test` as the element test predicate.

Function: `check%create-temporary-var`
Arguments:
    `type`
    `owner`
    `context`
Documentation: Creates a `desc$local` for a unique variable with type `type` and owner `owner` and inserts it into the locals of `context`. Returns a `cast$expr-var` corresponding to the local.

Function: `check$lisp-name-generator`
Arguments:
    `desc`
Documentation: Creates a lisp name for a desc.

Function: `check$set-lisp-name`
Arguments:
    `desc`
Documentation: Sets the lisp name of a desc.

Function: `check$has-lisp-code`
Arguments:
    `x`
Documentation: Checks if the given desc has lisp code created.

Function: `check$get-lisp-code`
Arguments:
    `x`
Documentation: Returns the lisp code of the given desc.

Function: `control$ensure-module-checked`
Arguments:
    `mod`
Documentation: Makes it shure that a module is checked.

Function: `desc$equal`

Arguments:
    a
    b
Documentation: Tests two descs for equality.

Function: `check%add-reference`
Arguments:
    from-constant
    to-constant
    &optional point
              reference-type
Documentation: Inserts a reference from `from-constant` to `to-constant`. `point` is the place of source where the reference was detected and `reference-type` the type of the reference.

Function: `check%add-reference-to-declarer-of`
Arguments:
    context
    named
    &optional point
              reference-type
Documentation: Inserts a reference from the constant corresponding to `context` to `named`. `point` and `reference-type` are same as in `check%add-reference`.

Function: `check%add-reference-to-module`
Arguments:
    context
    to-module
    &optional point
              reference-type
Documentation: Inserts a reference from the constant corresponding to `context` to the namer of `to-module`. `point` and `reference-type` are same as in `check%add-reference`.

Function: `check%declarer-of`
Arguments:
    context
    desc
Documentation: Returns the declarer of the given `desc`. `context` is not used.

Variable: `*check%module-name-string-limit*`
Initial value: 40
Documentation: This variable controls the length of the string that `check%module-name-string` returns. `NIL` means no limit.

Function: `check%module-name-string`

Arguments:

```
module
&optional context
```

Documentation: Returns a string that can be used as the name of the given desc.

11.3 Status

The module `status` defines the states that a constant descriptor may have and various functions that operate on states.

The states are

consistent The constant has passed checking with no errors and code has been generated for the constant.

errorneous Some errors were detected during the checking of the constant.

needs-checking The constant should be checked because its definition or something that it depends on has changed.

needs-codegeneration The constant is otherwise consistent, but code must be generated for it.

checking The checking of the constant is in progress.

codegenerating The generation of code for the constant is in progress.

deleted The constant has been deleted.

For every state there is a macro (with name `status$`*state*) that returns that state and a test predicate (with name `status$`*state*`-p`). There is also predicate `status-p` that returns T if the argument is any legal state and otherwise NIL.

For some states there are explicit sets that contain those constants that are in the corresponding states. These states are *needs-checking*, *needs-codegeneration*, and *erroneous*. Function `status$set-status` should be used for setting the state of a constant because it also updates the sets.

11.4 Checker Top Level

This module contains the top level function of the checker and some other functions.

Variable: `*check%queue*`
Initial value:

Documentation: The checker queue contains all constants that should be checked (see documentation of `queue`).

Function: `check$top-level-init`
Arguments:

Documentation: This function initializes the checker queue to an empty queue.

Function: `check$top-level`
Arguments:

Documentation: While there are constants in the checker queue this function applies `check%check-constant` to the first constant of the queue.

Function: `check%check-constant`
Arguments:

    constant
    &optional next-phase

Documentation: If `next-phase` is NIL the next phase of the checking of `constant` is performed. If `next-phase` is given then the checking of phases up to `next-phase` of constant performed. The constant is removed from the checker queue before checking and inserted to the checker queue after checking.

11.5 Check Module

This module contains functions and methods for the checking of module equates, i.e. equates whose right side is `past$proc`, `past$iter`, `past$rule`, `past$foreign-proc`, or `past$type-def`.

Macro: `past%routine-check`
Arguments:

    long-name
    short-name
    returns-or-yields

Documentation: This macro creates a check method for `past$proc`, `past$iter`, and `past$rule` if used in the following way:

```
(past%routine-check procedure proc returns)
(past%routine-check iterator iter yields)
(past%routine-check rule rule yields)
```

The arguments of the check method are:

context
constant
required-phase

`Context` is the context in which the checking should appear (the context of `constant`), `constant` is the constant corresponding the module that is checked, and `required-phase` is the phase up to which the checking must continue. As an implicit first argument `self` the method has of course the module PAST node.

This method is phase oriented (see the documentation of `check%module`). The phases are

:routine-start This is the initial phase. The constant corresponding to the routine is in this phase before checking begins.

:routine-init In this phase the routine descriptor is created if it did not already exist. The supermodule and datatype of the routine are computed and the submodule list is initialized to NIL. The outermost scope of the routine is entered. The equate select and the parameters — if present — are checked. The instance list is initialized to NIL if this is a generator.

:routine-header The arguments, the return or yield types, and the signals of the routine are checked. The type of the routine is computed.

:routine-wheres The where restrictions are checked.

:routine-body The owns, binds and statements of the routine are checked. The outermost scope of the routine is exited and the constant status is set to `status$needs-codegeneration`.

Check method for: `past$type-def`
Arguments:
`context`
`constant`
`required-phase`
Documentation: This method is phase oriented (see the documentation of `check%module`). The phases are:

:datatype-start The initial phase. The constant corresponding the datatype is in this phase before the checking begins.

:datatype-init The datatype descriptor is created if it did not already exist. The supermodule and the type fields are set. The outermost scope of the datatype is entered. The equate select and the parameters — if present — are checked. The instance list is initialized to NIL if this is a generator.

:datatype-wheres The where restrictions of the datatype are checked.

:datatype-rep The representation type specification of the datatype is checked.

:datatype-routines The datatype `is` and `inherits` lists are checked.

:datatype-body The datatype owns are checked. The outermost scope of the datatype is exited and the constant status is set to `status$needs-codegeneration`.

Check method for: `past$foreign-proc`
Arguments:
`context`
`constant`
`required-phase`
Documentation: This method is phase oriented (see the documentation of `check%module`). The phases are

:routine-start This is the initial phase. The constant corresponding the foreign proc is in this phase before the checking begins.

:routine-init In this phase the foreign proc descriptor is created if it did not already exist. The supermodule and datatype of the foreign proc are computed and submodule list is initialized to NIL. The outermost scope of the foreign proc is entered. The equate select — if present — is checked. The foreign language is checked.

:routine-header The arguments and the return types of the foreign proc are checked. The type of the foreign proc is computed.

:routine-wheres The where restrictions are checked.

:routine-body The owns, binds, statements, etc. are set to NIL. The outermost scope of the foreign proc is exited and the constant status is set to `status$needs-codegeneration`.

Function: `check%check-language`
Arguments:
`lang`
Documentation: Checks that the language is either LISP or PLM.

Function: `check%check-equate-select`
Arguments:
`context`
`constant`
`field-name-idn`
`field-type-idn`
`field-list-idn`
Documentation: Checks that the `foreach` clause is legal.

Macro: `define-parms-selector-check`

Arguments:

`ordered-or-unordered`

Documentation: Creates check methods for `past$parms-selector-ordered` or `past$parms-selector-unordered` according to the argument `ordered-or-unordered`. The check method has only one argument `context`. It checks that there are no name conflicts and then creates a `desc$field-list` and returns it.

Check method for: `past$parms-normal`

Arguments:

`context`

Documentation: Checks a normal parameter list. Creates `desc$parameters` for all parameters and inserts them into the `context`. The actual checking is done by `check%check-parm`.

Function: `check%check-parm`

Arguments:

`pd`

Documentation: Checks that the parameter type is a legal parameter type.

Function: `check%check-arg`

Arguments:

`decl`
`context`

Documentation: Checks an argument declaration and returns two values, a list of `desc$args` and a list of type descriptors (the external types of the arguments).

Function: `check%check-args`

Arguments:

`decls`
`context`

Documentation: Checks a list of argument declarations using `check%check-arg`.

Function: `check%check-type-spec-cvt-list`

Arguments:

`type-specs`
`context`

Documentation: Checks a list of type specifications and returns two values, the external and the internal type descriptors. The type specifications may contain `cvt` in which case the external and internal types are different.

Function: `check%check-signals`

Arguments:

`signals`
`context`

Documentation: Checks the signals of a routine. The types of the signals are checked and the names of the signals are checked for multiple definitions of the same name. `desc$signal` descriptors are created for the signals and the signals are inserted into the `context`. Two values are returned, a list of external and internal signal descriptors.

Function: `check%check-binds`
Arguments:
    `idn-list`
    `context`
Documentation: Checks a binds list. The idns are checked against multiple definitions of the same name. It is also checked that the idns name a bindable variable. The variables are inserted into `context`.

Function: `check%check-decls`
Arguments:
    `decls`
    `context`
Documentation: Checks a list of declarations and returns two values, a list of idns and a list of type descriptors. The checking of a single declaration is performed by the check method of `past$decl`.

Function: `check%check-own`
Arguments:
    `past`
    `context`
Documentation: Checks an own definition. The declaration part is checked and then the assignment of the initialization part to the declaration is checked. Then `desc$owns` are created for the owns and they are inserted into the `context`. Two values are returned, a list of the own descriptors and a statement cast for the initialization of the owns or NIL if there was not initial value.

Function: `check%check-owns`
Arguments:
    `own-pasts`
    `context`
Documentation: Checks a list of own definitions and returns two values, the list of own descriptors and a `cast` for the initialization of the owns. The actual checking is performed by `check%check-own`.

Function: `check%check-restriction-parameter`
Arguments:
    `idn`
    `foreach`
    `context`

Documentation: Checks the parameter that is used in a where clause. If `foreach` is given it is also checked.

Check method for: `past$restriction-has`
Arguments:
    `context`
Documentation: Checks a where clause of the `has` type. The required routines are checked and `desc$required-routine`s and `desc$constant`s that have the required routine descriptors as their values ar created. The constants are returned. If the restriction has a `foreach` definition the checking takes place inside a new scope.

Check method for: `past$restriction-in`
Arguments:
    `context`
Documentation: Checks a where clause of the `in` type. The typeset specification is checked and required routines are checked and `desc$required-routine`s and `desc$constant`s that have the required routine descriptors as their values ar created. This is done with `check%substitute-typeset`. The constants are returned. If the restriction has a `foreach` definition the checking takes place inside a new scope.

Function: `check%substitute-typeset`
Arguments:
    `typeset`
    `parm`
    `flist`
    `context`
Documentation: For every required routine of the typeset a `desc$required-routine` and `desc$constant` that has the `desc$required-routine` as its value. The constants are required.

Function: `check%check-rep`
Arguments:
    `constant`
    `context`
    `beg-point`
Documentation: Checks that the datatype has a representation type defined and it is a legal type.

Function: `check%check-is`
Arguments:
    `op-names`
    `equates`
    `constant`
    `context`

Documentation: Checks an `is op-names` list. The operation names should match routines that are given in `equates`. Returns three values, the constants of the routines corresponding `op-names`, the points of the `op-names` (these are used when giving error messages) and a flag that indicates whether errors were detected.

Function: `check%check-is-oper-name-idn`
Arguments:

```
node
context
equates
constant
```

Documentation: Checks a normal operation name in the `is` list. It should match an equate in `equates`.

Function: `check%check-is-oper-name-select`
Arguments:

```
node
context
equates
constant
```

Documentation: Checks an operation name with a field list in the `is` list. It should match an equate in `equates`.

Function: `check%find-type-routine`
Arguments:

```
name
equates
constant
point
context
```

Documentation: Finds a routine with `name` in the `context`. There should also be an equate in `equates` with the same name.

Function: `check%check-inherits`
Arguments:

```
op-names
rep
context
```

Documentation: Checks that the `inherit op-names` are legal operation names that match operations of the `rep` type. Returns three values, the constants of the routines corresponding `op-names`, the points of the `op-names` (these are used when giving error messages) and a flag that indicates whether errors were detected.

Function: `check%check-inherits-oper-name-idn`
Arguments:

```
node
rep
context
```

Documentation: Checks a normal operation name list in the `inherits` list. It should match a routine of the `rep` type.

Function: `check%check-inherits-oper-name-select`
Arguments:
`node`
`rep`
`context`
Documentation: Checks an operation name with a field list in the `inherits` list. It should match a routine of the `rep` type.

Function: `check%find-inherited-routine`
Arguments:
`name`
`rep`
`context`
Documentation: Tries to find a routine of the `rep` type with name `name`. Creates a `desc$inherited-routine` or `desc$inherited-routine-generator` for the routine and a constant that has the inherited routine as its value. Returns the constant.

Function: `check%check-routines`
Arguments:
`constants`
`points`
`context`
Documentation: If we are checking an instance, simply returns `constants`. Otherwise loops through `constants` and verifies that there are no two constants with the same name. The constants are divided into two groups, those that have a `foreach` (select) and those that do not have it. Then it is checked that there is no constant with no `foreach` that has a name that is the prefix of a name of a constant that has `foreach`. Returns two values, the constants and an error flag.

11.6 Refs

The module `refs` defines the type `reference` — references from constants to other constants. A reference consists of fields `from` (the referring constant), `to` (the referred constant), and a list of reference type and point pairs. The reference types are `:header`, `:body`, `:bind`, `:inline`, and `:is`.

11.7 Assignment Checker

The assignment checker consists of module *assign-check*, which defines a check method for `past$statement-assign` nodes. The method needs two arguments: the PAST to be checked and the context where it is checked. (The reader should note that all other assignments, except for the assigment statement, are checked by the expression checker.)

The left side of an assignment statement is a list of references and the right side is a list of expressions or a single invocation returning a matching number of values. At execution the values of the expressions (or the invocation) are assigned to the slots referred to by the references.

A reference is a variable, or an expression that has an indexing or component selecting qualifier attached to it. (e.g., `person.age`). The latter case is transformed into a update invocation.

An indexing qualifier is transformed to an invocation of the `store` operation of the type of the variable:

a[i] := b → type_of_a$store(a, i, b)

A compoment selecting qualifier is transformed to an invocation of the `set_` operation for the component:

a.c := b → type_of_a$set_c(a, b)

The qualified expression accesses the object that is to be updated. For example

first(:person.siblings).age := x

is transformed to

age_type$set_age(person_type$first(person_type$siblings(person)), x)

The assignment check method first constructs the left hand sides. For each variable on the left hand side a `cast$expr-var` is created. For each expression qualified by an index on left hand side a `cast$expr-invocation` node for the `store` operation is created. The first argument of the `cast$expr-invocation` is set to be the CAST of the expression qualified and the second argument is set to be the CAST of the index. The third argument is left `NIL`. For each expression qualified by a component name on left hand side a `cast$expr-invocation` node for the `get_` operation is created. The first argument of the `cast$expr-invocation` is set to be the CAST of the expression qualified. The second argument is left `NIL`.

After construction of the left hand side the right hand side is constructed. If there is a single variable on the left hand side then a `cast$statement-assign` is created.

If there is a single qualified expression on the left hand side then the argument previously left `NIL` is set to be the CAST of the right hand side. This is done by the function `check%create-assignment-cast`.

If there are several references on the left hand side then a more complex CAST must be build. At execution the index expressions on the left hand side must be evaluated before the right hand sides, and they must be evaluated before the update operations are performed. Therefore the index expressions and right hand sides are assigned to temporary variables that are set to be the corresponding arguments of the update invocation CASTs. This is done by using the function `check%create-splitted-assignment-casts` . The result is a sequence of assigment and invocation CASTs wrapped in a `cast$statement-begin-end` .

12.8 The Statement Checker

The statement checker transforms the PAST nodes of type statement to corresponding CAST nodes and performs the checks listed in XE Reference Manual. The statement checker is defined as a PAST method using the *defcheck* environment. The checks have arguments `self` and `context`. As many statements in XE-language contain scopes, the scope handling is an important part of the statement checker.

Many constructions of the XE language can be seen as a general assignment, where a list of expressions must match a list of types in a given context. Therefore a general function called *check%check-general-assignment* is defined to perform the checks involved.

General assignment

Checks that the given list of expressions matches the given list of types. Two possibilities exists: either there are as many expressions as there are types, or there is only one expression, which is an invocation. The invocation is checked with *check%check-invocation*. The error notifications given are: *no-values-expected, too-many-values, missing-values*. The function returns the result of the expression checks.

past$statement-begin-end

Returns the value of the body check.

past$body

Checks `statement list` in scope `past$body-scope` using equates `past$body-equates`. Returns a `cast$statement-begin-end` with the checked statements.

past$statement-break

Searches from context the enclosing loop. If none is found notifies a `homeless-break` otherwise returns a `cast$statement-break` referring to the loop found.

past$statement-case

Checks `expression`, checks the list of `options`, and checks the `otherwise`. If no errors are found, checks that all `options` are different and notifies `duplicate-case-expr`, if they are not. Returns a `cast$statement-case` referring to the checked `expr`, `options`, and `otherwise`.

past$case-option

Checks in the scope of the `past$case-option-body` using its equates `expressions` and `statement list`. Checks that the `expressions` are constant, notifies `constant-expected` if they are not. Returns a `cast$case-option` referring to the checked expressions and to the checked statements. The statement list is inserted in to a `cast$statement-begin-end`.

past$statement-continue

Searches from `context` the enclosing loop. If none is found notifies a `homeless-continue` otherwise returns a `cast$statement-continue` referring to the loop found.

past$statement-decl

Checks the `decls`, and then checks `expressions` using the general assignment check. Then a `desc$locals` are created, and inserted to `context`. If a `name conflict` is found it is notified. According to the number of expressions either a `cast$statement-noop, cast$statement-assign, or` a `cast$statement-begin-end`, having a list of assignments in its statement list is returned.

past$statement-exception-handler

The checking order of statement-exception-handler is a bit complicated. First `equates` and `declarations` of all `handlers` are checked, then a new handler scope is created and handlers inserted into it, and `statement`, around which the handler is wrapped is checked. After the main `statement` has been checked the scopes of bodies of the `handlers` are reentered and `statements` in them checked. The separation of header and body checks is done using the macros `define-header-check` and `define-body-check`. The check returns `cast$statement-except` or `cast$statement-resignal`.

past$statement-exit

Checks that `name` to exit is a signal name, that there is a handler for it, and that the types of `values` signaled match the definition of the signal. Error notifications are accordingly:`exit-failure`, `no-handler-for-exit`, and the messages reported by general assignment check. Returns a cast$statement-begin-end containing the assignments to the values, and a cast$statement-exit.

past$statement-for

Creates a new scope for the loop body, and checks `declarations` in it. Creates `desc$iter-local` variables and checks the assignment compatibility between `iterator invocation` and the variables. Checks that their is no name conflict and creates a `cast$statement-for`. The `body` is then checked within this statement to allow the references to the loop structure within the body and `cast$statement-for` is returned.

past$statement-if

Checks `condition-action` pairs and `else body`. Returns a `cast$statement-if`.

past$cond-act

Checks the condition action pair in the scope of the action body and returns a `cast$cond-act`. Uses expression checker to check that the condition is of type `bool`.

past$statement-invocation

Checks the invocation using `check%check-invocation` and returns `cast$statement-invocation`.

past$iterate

For each `iterate` checks `declarations` and creates `desc$iter-locals` for them. Checks that `iteration invocation` matches the declaration list, and checks that it is an iterator invocation notifying `illegal-iterator-invocation` is it is not. Creates a `cast$iteration` for each iteration. Inserts the declared variables into `context` and notifies `name-conflict` if a conflict is found. Inserts the iterations into `context` and notifies `name-conflict` if a conflict is found between the iteration labels. Checks `statements` inside iteration, creates a `cast$statement-iterate`, and returns the `cast$statement-iterate`.

past$statement-next

Searches from `context` the enclosing iteration named according to the `label`. If none is found notifies a `no-such-iteration` otherwise returns a `cast$statement-next` referring to the iteration found.

past$statement-return

Finds the enclosing module. In none is found notifies `return-outside-procedure`, otherwise checks the assignment compatibility of `return values` and returns a `cast$statement-return`.

past$statement-signal

Finds `signal` and notifies `undeclared-signal` if none is found. Checks signaled `values`, and the assignment compatibility of the signaled values and the types in the declaration, and returns `cast$statement-signal`.

past$statement-static

Returns a `cast$statement-static`.

past$statement-tagcase

The transformation is done using the desugared version of tagcase. This routine does only the additional checks required. It checks that the type has routine `tag_of`, all tags are known, a given tag occurs only once, and that all tags are used.

past$statement-while

Opens a scope for `while body`. Checks `equates`, checks the test `expression` requiring that it has to be boolean, creates `cast$statement-while`, checks `statements` inside it, and returns it.

past$statement-yield

Checks that the surrounding module is an iterator or rule, otherwise notifies `homeless-yield`. Checks `values` yielded, and that the values are assignment compatible with the values specified in the module header. Creates a cast$statement-yield.

13.9 Instance Checker

Routine and data type generators are *instantiated* to create modules that can be used like normal non-parameterized modules. A generator is provided with actual constant values for the formal parameters declared in its header. Legality of the actual parameters must be checked. This is done by the *instance checker*, which consists of a single module: *check-instance*.

Instances are checked by the function:

Function: `check%check-instance-restrictions`
Arguments:
    `desc`
Documentation: `desc` must be a `desc$instance`. The function checks that the actual parameters match the restrictions defined in the generator of the instance. If errors are found error messages are given and an error node is returned.

`Check%check-instance-restrictions` is only an interface that calls the actual checking functions to perform the check. The checking of ordinary and selector generator instances are performed by separate functions.

Function: `check%check-ordinary-instantiation`
Arguments:
    `instance`
    `generator`
    `parameter-values`
Documentation: `Instance` must be the instance with the `parameter-values` of the `generator` . `generator` must be an ordinary generator. The function checks that the actual parameters match the restrictions defined in the generator of the instance. If errors are found error messages are given and an error node is returned.

The parameter values that are not types need no checking, because such formal parameters cannot have restrictions. For type parameters the primary and secondary restrictions are collected from the generator and the **use** definitions from the parameter values. The routine types in the restriction are "pseudo-types" because they contain formal type parameters. Therefore the formal parameters in the routine types of the restrictions are substituted by the actual types. (This is performed by use of `desc$type-subst`.) If use definition is not found for a restriction then the actual parameter type is checked to have one. After this the routine type of the actual routine is checked to be assignment compatible with the substituted routine type of the corresponding restriction.

Absence of a routine required by secondary restriction does not make the whole data type unusable, only the operation that declares the restriction cannot be used. Therefore the instance checker does not give an error message if a routine for a secondary restriction is missing.

Function: `check%check-selector-instantiation`

Arguments:

`instance`
`generator`
`actual-field-list`

Documentation: `Instance` must be the instance with the `actual-field-list` of the `generator` . The generator must be a selector generator. The function checks that the actual parameters match the restrictions defined in the generator of the instance. If errors are found error messages are given and an error node is returned.

Besides the above functions there are some auxiliary functions defined in the module:

Function: `check%get-primary-restrictions`

Arguments:

`generator`
`parameter`

Documentation: Returns a list of primary restrictions (of type `desc$required-routine`) for the formal parameter `parameter` of `generator`. The generator must be an ordinary generator.

Function: `check%get-secondary-restrictions`

Arguments:

`generator`
`parameter`

Documentation: Returns a list of secondary restrictions (of type `desc$required-routine`) for the formal parameter `parameter` of `generator`. The generator must be an ordinary generator.

Function: `check%get-primary-restrictions`

Arguments:

`generator`
`parameter`

Documentation: Returns a list of primary restrictions (of type `desc$required-routine`) for the formal parameter `parameter` of `generator`. The generator must be a selector generator.

Function: `check%get-secondary-restrictions`

Arguments:

`generator`
`parameter`

Documentation: Returns a list of secondary restrictions (of type `desc$required-routine`) for the formal parameter `parameter` of `generator`. The generator must be a selector generator.

14.10 The Expression Checker

The expression checker checks the legality of any PAST consisting of options defined in the `past$expr` domain and builds the corresponding CAST representation and descriptors. If the PAST given to the expression checker is found illegal then appropriate error messages are given.

The expression checker is defined in modules *expr-check* and *expr-check-aux*. The module expr-check contains definitions of check methods for each node type in domain `past$expr`. The module expr-check-aux contains a set of auxiliary functions needed in the checking of expressions.

The functions defined in expr-check-aux are the following:

Function: `check%get-operation-of-type`
Arguments:

    context
    type-desc
    opname

Documentation: Returns the descriptor of operation *opname* of type *type-desc* in the given *context*. The context is needed because the operations of parameters (declared by **where** clauses) are stored in the context. The *opname* must a string (in upper case letters) and *type-desc* must be a `desc$type`.

Function: `check%get-all-operations-of-type`
Arguments:

    context
    type-desc

Documentation: Works as `check%get-operation-of-type` , but returns a list of all operations.

Function: `check%get-result-types`
Arguments:

    routine-type

Documentation: Returns a list of result types of *routine-type*.

Function: `type$match`
Arguments:

    object-type
    slot-type

Documentation: Returns `NIL` if *object-type* is not assignment compatible with *slot-type* as defined in the /XE Reference Manual/.

Function: `check%proctype-matches`
Arguments:

    desc
    args
    results

Documentation: Returns `NIL` if the *desc* is not a `desc$proctype` or if it is not assignment compatible with type **proctype**(*args*) **returns**(*results*) **signals**(*) as defined in the /XE Reference Manual/.

Function: `check%signal-handling`
Arguments:

    context
    signal-desc-list
    point

Documentation: Checks that the signal value type of the signals in *signal-desc-list* are assignment compatible with their handlers found in *context* (if they are defined). Returns `NIL` if an illegal handler is found and attaches an error message to the source code location *point*.

Function: `check%constant-proc-p`
Arguments:

    routine-desc

Documentation: Returns `NIL` if *routine-desc* cannot be used in constant expressions. (The constant procedures are defined in /XE Reference Manual/ and /Notes on Code Generation/).

Function: `check%parameters-match`
Arguments:

    parameter-list
    type-list

Documentation: Returns `NIL` if the parameters in *parameters* cannot be assigned to formal parameters of types given in *type-list*.

Function: `check%constant-p`
Arguments:

    routine-desc
    routine-args

Documentation: Returns `NIL` if *routine-desc* applied to *routine-args* does not result in a constant value.

Function: `check%use-conflicts`
Arguments:

    context
    param-values

Documentation: Returns `NIL` if different use definitions are given for the same type in *param-values*. (Gives the error messages needed using the points stored in the *param-values*.)

The expression checker itself consists of check method definitions for each PAST node type. Methods for the nodes that resemble each other are implemented using a macro definition.

All the check methods have three arguments: the node to be checked, the context where the check is to be made, and the *required type*. The required type is needed because of the literal and operator overloading of XE. The required type can have four kinds of values. If it is the type descriptor of an XE type T then the type of the expression to be checked is expected to be assignment compatible with T (messages are given for the errors detected). If it is the keyword `:deduce-type` then the type of the expression is deduced (an error message is given if the deduction is impossible). If it is the keyword `:peek-type` then type of the expression is deduced, but no error messages are given even if an error occurres (an error node is returned to inform the caller). If it is the keyword `:field-list` then the expression is assumed to be a field list. The first three options are handled by any expression check method, but the last one only by methods for struct constructors and field lists.

To understand how the required type is used assume that the statement

```
a:  byte := 1 + 2
```

is to be checked. The expression checker is called to check the PAST for `1 + 2`. Because the result of the expression is to be assigned to `a` the type of the expression must be `byte`. The check method for invocations is given `byte` as the required type and it passes the type requirement to the check method for integer literals, because the argument and result types for the `+` operation are equal. For both literals the check method for integer literals observes that the type `byte` has an overloading routine for integer literals, and thus `1` and `2` can represent byte objects.

If an equate `a = 1 + 2` is to be checked no type is declared for `a`. Therefore the check method for invocations is given the argument `:deduce-type`. It performs the type deduction by first calling the check method for literal `1` with the argument `:peek-type`. Because of the literal overloading, it is impossible to say what is the type of the object represented by literal `1`. The integer literal method does not give an error message, but it returns an error node. After this the check method for invocations calls the method for checking literal 2 with the argument `:deduce-type` , because the right subexpression is now free to decide the type of the whole expression. The check method for integer literals returns the default type `int`, because no type is explicitly defined. (Actually the method returns a CAST for the literal – the type contained in the auxiliary type field of expressions.) The check method for the left subexpression is called again (with `int` as required type), because the type of the expression is now known.

At first glance the type deduction mechanism of the expression checker may seem rather complicated. However, it is fast enough and acts as specified in the /XE Reference Manual/. The reader should notice that most of the routines handle the required type in a straight forward manner (e.g., passes it to an other check routine). The actual "thinking" is done by the invocation and literal check methods.

Usually a check method first collects data needed for checking, applies the context rules (and gives error messages if necessary), constructs the result object and returns it. Typically a CAST is returned, but some methods return a descriptor. Sometimes a node can have several interpretations or forms. Methods for these nodes have several branches.

The check methods that work in a straight forward manner and return a CAST node corresponding to the PAST node given are:

```
past$expr-true
past$expr-false
past$expr-up
past$expr-down
past$expr-terminated
past$expr-cand
past$expr-cor
past$expr-typed-operation
past$expr-selector-operation
```

The check methods that work in a straight forward manner and return a descriptor corresponding to the PAST node given are:

```
past$expr-proc
past$expr-iter
past$expr-rule
past$expr-foreign-proc
past$expr-type-def
past$expr-field-list
past$expr-field-list-idn
past$expr-typeset-spec
past$expr-proctype
past$expr-ruletype
past$expr-itertype
past$expr-type
```

The method `past$expr-parenthesis` passes all values and does no checking. The method for `past$expr-cvt` always gives an error message, because **cvt** is allowed only in the headers of data type operations and the module checker handles these cases.

The check methods for the simple literals

```
past$expr-int-literal
past$expr-char-literal
past$expr-string-literal
```

are implemented using a common macro `check%define-simple-overloaded-literal`. The check methods produced by the macro first determine the type of the literal and then check that the type has the overloading operation needed. If overloading is necessary (i.e., the type is not a built-in type) then a CAST node for the invocation of the operation is created.

The check method for `past$expr-tag-literal` works in a quite similar way, but it does not use any default types for the literals, because no such types are specified in the XE language. A CAST node for the invocation corresponding to the tag literal is always built.

The check method for `past$expr-sequence-cons` works like the checks for simple literals, but the type deduction (`:deduce-type`) is more complex. For example, assume the equate `a = [1, 2, byte$3]` to be checked. The elements of the constructor are peeked one after another until an element that has a fixed type is found (in the example the last one). After this all the other elements are overloaded to that type and a sequence of that type is constructed. If no element has a fixed type and (`:deduce-type`) is used then a sequence of the default type of the first element is constructed.

The checks for `past$expr-struct-cons` and `past$expr-field-list-list` are the most complicated ones. The check method for `past$expr-struct-literal` directly uses the method for `past$expr-field-list-list` because the PASTs are similar. The latter method first checks that there are no duplicated fields specified. After this it tries to solve the type of the literal. If the required type is `:field-list` or a type descriptor then a field list or a struct is built, respectively. (If the type descriptor is not an instance of the struct generator then the overloading is handled as for simple literals.) If the required type is `:peek-type` then an error has occurred, because the type of the literal is not known. Otherwise the type is deduced from the values of the fields. If the values are types then the literal is a field list, if not, it is a struct.

The legality of the fields of a field list is checked by `check%deduce-field-list` , which returns a field list descriptor. The function `check%deduce-struct-cons` checks fields of a struct and returns a `cast$struct-cons` for the struct type. Both functions have two arguments: the context and the field list node.

The check method for identifiers (`past$expr-idn`) fetches the descriptor of the identifier from the context and determines if it is legal to use the identifier. The method does not always return the descriptor, but sometimes wraps it in a CAST node.

The check method for `past$expr-dot` handles the sugared accessing operations of structs, and records etc. It returns a `cast$expr-invocation` node for the invocation of the corresponding "GET_" operation.

The check method for `past$expr-brackets` checks both the sugared fetch operations and ordinary instances, whose syntactical outlook can be similar. The choice based on the identifier before the brackets. Assume, for example, that `a[b]` is to be checked. If the identifier `a` refers to a generator then the expression is interpreted as an instance of the generator and the descriptor of the instance is returned. (Note that routine descriptors are wrapped in a `cast$expr-routine` node.) Otherwise it is assumed to be a sugared fetch operation and the corresponding `cast$expr-invocation` is returned.

The check method for `past$expr-selector-instance` is implemented by using the `past$field-list` check method to handle the actual parameter list. Thus the method is quite simple, it calls directly the instantiator to create the instance.

The `past$expr-invocation` nodes are handled by a rather long method, because there are many forms of invocations and type deduction is involved in the sugared ones. The type deduction is done for each kind of an invocation in a way similar to the type deduction examples above. The result of the check is a `cast$expr-invocation` node.

Iterator invocations and the invocations that have several result values are not handled by the check method for `past$expr-invocation` nodes, but by the function:

Function: `check%check-invocation`
Arguments:

    node
    context
    proc-p
    required-types

Documentation: *Node* is the past node to be checked. *Context* is the current context and *proc-p* is a flag that must be `NIL` if an iterator or a rule is expected. The *required-types* must be a list of expected result types that is `NIL` if results are not used by the invoker. No keyword values (e.g., `:deduce-type`) are allowed.

The `past$param-value` nodes are not checked by the check method for the node, but by the function:

Function: `check%past-param-value`
Arguments:

    self
    context
    formal-param-type

Documentation: *Self* is the past node to be checked. *Context* is the context where the instance is made. (The function makes a new scope for the checking of the possible **use** declarations.) *Formal-param-type* is the type descriptor for the corresponding formal parameter. The function returns a `desc$parameter-value` descriptor.

Besides the checking actions explained, every check method (and the few special check functions) are able to handle errors. If an error is found in an XE expression then the checking of the *whole* expression is aborted resulting in a `past$error` node. The error handling is done using the facilities common for all checking routines of the XE compiler.

The expression checker does not do any evaluation of the XE expressions, except for folding the tag expressions (i.e., evaluates the calls for built-in tag type creators to the corresponding tag values). This is done by calling `eval$tag-folder` when CAST nodes for invocations are created.

16 The Smart Recompilation Package

The smart recompilation package consists of three modules. `Pdiff` contains the functions that compute differences between two sets of equates (`pasts`). `Affected` contains the functions that compute the changed and affected constants when given the equate changes that are computed in `pdiff`. It also contains a function that modifies these constants according the type of the modification. `Scope-copy` contains a structural method for copying scopes.

16.1 Pdiff

This module contains facilities for computing differences between sets of `past` trees.

Struct: `equate-changes`
Fields:

`parent`	*equate or nil*
`not-modified`	*list(equate)*
`add`	*list(equate)*
`delete`	*list(equate)*
`modify`	*list(modified)*

Documentation: Records differences between two sets of `pasts` *old* and *new*. `parent` is either `nil` or a `past` node that contains these equates. `not-modified` contains those equates that were both in *new* and *old*. `add` contains those equates that were in *new* but not in *old*. `delete` contains those equates that were in *old* but not in *new*. `modify` contains a list of `modified` structs corresponding to modified equates.

Struct: `modified`
Fields:

`new-equate`	*past$equate*
`old-equate`	*past$equate*
`mod-types`	*list*(`:module-type or :header or :body or :is or equate-changes`)

Documentation: This struct describes a modified equate. *mod-types* is a list that describes the modifications. `:module-type` means that the type of the descriptor corresponding the module must be changed. `:header`, `:body`, and `:is` mean that the type of the module has not changed, but the header, body, or `is` list of the module has changed. `equate-changes` is a struct of type `equate-changes` and it means that the equates in the beginning of the module have changed.

Function: `pdiff$calculate-equate-changes`
Arguments:

`old-equates`
`new-equates`
`&optional parent`

Documentation: Computes the difference between two sets of `past$equates` `new-equates` and `old-equates`. Returns NIL if no changes were found and a `equate-changes` struct if changes were found.

Function: `pdiff$calculate-modified`
Arguments:
    `old-equate`
    `new-equate`
Documentation: Computes the differences between two equates with the same name and returns a `modified` struct if they are different and NIL if they are equal.

Form: `(define-abstract-syntax-generic-function past%calculate-modified past)`
Documentation: A generic function that is used in the computation of differences between two `past` nodes. The methods for this generic function are defined only for `past$exprs` and `past$proc`, `past$iter`, `past$rule`, `past$foreign-proc`, and `past$type-def`.

Past method: `past%calculate-modified` of `past$expr`
Fields:
    `new`
    `old-equate`
    `new-equate`
Documentation: A default method for `past$exprs`. If the domain options of `self` and `new` are different a `modified` struct is returned with modification type `:module-type`. If the option is the same and the nodes are not equal, a `modified` struct is returned with modification type `:header`. Otherwise NIL is returned.

Macro: `past$expr-module-calculate-modified`
Arguments:
    `module`
Documentation: A macro that creates a `past%calculate-modified` method for `past$expr-proc`, `past$expr-iter`, `past$expr-rule`, `past$expr-foreign-proc`, or `past$expr-type-def` depending on the argument `module`. The method has three arguments `new`, `old-equate` and `new-equate`. The method returns a `modified` struct with modification type `:module-type` if `self` and the `new` are not of the same option. Otherwise it calls `past%calculate-modified` for the inner field of the `pasts`.

Macro: `past$routine-calculate-modified`
Arguments:
    `option`
    `returns-or-yields`
Documentation: A macro that creates a `past%calculate-modified` method for `past$proc`, `past$iter`, and `past$rule` depending on the argument `option`. The

method has three arguments `new`, `old-equate` and `new-equate`. The method returns a `modified` struct with modification type `:module-type` if `self` and the `new` are not of the same option. If they are of the same option, but `self` has parameters and `new` does not or vice versa a `modified` struct with modification type `:module-type` is returned. Otherwise if the parameters, argument types, return or yield types, signals, or wheres are not equal a `modified` struct with modification type `:header` is returned. Otherwise if the argument names, the binds, the owns, the statements, or the conditions of the rules are not equal a `modified` struct with modification type `:body` is returned. If the equates in the beginning of the module are changed the `equate-changes` struct is included in the modification types of the `modified` struct. If no changes are found NIL is returned.

Past method: `past%calculate-modified` of `past$foreign-proc`
Fields:

`new`
`old-equate`
`new-equate`

Documentation: Calculates modifications between `self` and `new`. Rather similar to the `past%calculate-modified` method for `past$proc` described above.

Past method: `past%calculate-modified` of `past$type-def`
Fields:

`new`
`old-equate`
`new-equate`

Documentation: Calculates modifications between `self` and `new`. Rather similar to the `past%calculate-modified` method for `past$proc` described above.

Function: `pdiff%similar-signals`
Arguments:

`s1`
`s2`

Documentation: Tests the similarity of two signals clauses and returns T if they are similar and NIL if they are not similar.

Function: `pdiff%similar-wheres`
Arguments:

`w1`
`w2`

Documentation: Tests the similarity of two lists of where clauses and returns T if they are similar and NIL if they are not similar.

16.2 Scope Copy

Defines a structural method `past%copy-scopes` with one argument `other` that copies the `&aux` fields with name `scope` from `self` to `other`.

16.3 Affected

This module contains the routines that collect information about affected constants with given `equate-changes`. The result of the computation is a set of `recompilation-items`.

Struct: `recompile-item`
Fields:

`constant`	
`new-def`	
`modification-type`	*:new, :deleted, :header-changed, :body-changed, :is-changed, :equates-changed, :module-type-changed, :header-affected, :body-affected, :code-affected*
`context`	
`scope`	

Documentation: This struct describes a constant that has to be recompiled. `constant` is the constant (if it already exists). `new-def` is the new `past` (if any). `modification-type` describes how `constant` has changed. `context` is the context and `scope` the scope that contain `constant`.

Variable: `*pc$deleted-scopes*`
Initial value:
Documentation: Those scopes that should be deleted are inserted into this list.

Variable: `*pc$changed-items*`
Initial value:
Documentation: Those `recompile-items` that are created during the computation of affected constants are inserted into this list.

Constant:`*pc%modification-type-aalist*`
Documentation: An association list that is used in the detection of the new modification type of a `recompile-item`. Every element of the list is of form `(`*old-modification-type . new-modification-type-alist*`)`. *new-modification-type-alist* is an association list that contains elements of form `(`*modification-type change-type-p propagate-p*`)`. When `rci$create` is called and a `recompile-item` with the same constant already exists we search `*past%modification-type-aalist*` for an element using the modification type of the existing `recompilation-item` as the search key. Then we search the *new-modification-type-alist* of that element for an element with the same *modification-type* that is passed to `rci$create` as an argument. If no such element is found an internal error has occured. Otherwise if the *change-type-p* field of the element is T the modification type of the `recompilation-item` is changed and if the *propagate-p* field of the element is T the element is pushed into the list
`pc$changed-items`.

Function: `rci$create`
Arguments:

```
&key constant
     new-def
     modification-type
     context
     scope
```

Documentation: If `constant` is NIL or it is non-NIL and it does not already have a `recompilation-item` `rci$create` creates one and inserts it into `*pc$changed-items`. Otherwise the modification type of the `recompilation-item` of `constant` and the list `*pc$changed-items*` are updated (see the documentation of `*pc%modification-type-aalist*`).

Function: `pc$propagate-changes`
Arguments:

```
ec
&optional (context *context*)
          (scope *context$top-level-name-scope*)
```

Documentation: First the changed items are collected. Then the changes are propagated. Then all created recompilation items and all deleted scopes are returned.

Function: `pc%collect-changed-items`
Arguments:

```
ec
context
scope
```

Documentation: Calls `pc%collect-changes-new`, `pc%collect-changes-deleted`, `pc%collect-modified`, and `pc%mark-equate-scopes` in order to collect all changed items.

Function: `pc%collect-changes-new`
Arguments:

```
equates
context
scope
```

Documentation: For every (new) equate in `equates` a `recompilation-item` is created.

Function: `pc%collect-changes-deleted`
Arguments:

```
equates
context
scope
```

Documentation: Creates a `recompilation-item` for every (deleted) equate in `equates`.

Function: `pc%collect-changes-modified`

Arguments:

`modifications`
`ignore`
`scope`

Documentation: Creates a `recompilation-item` for every `modified` struct in `modifications`.

Function: `pc%mark-equate-scopes`
Arguments:

`containing-scope`
`equates`

Documentation: Sets the `dont-delete-p` flags of the `first-name-scopes` of the contexts of all constants corresponding `equates`.

Function: `pc%collect-deleted-constants-inside`
Arguments:

`context`
`scope`

Documentation: Creates `recompilation-items` with modification type `:deleted` for every constant in `scope` and in children scopes of `scope`.

Function: `pc%propagate-changes`
Arguments:

Documentation: Computes the set of affected constants. The result is a set of recompilation items. In the beginning the set contains the recompilation items that `pc%collect-changed-items` has computed. Then these items are considered. An item is selected and if the modification type is not `:equates-changed` then all references to the constant of the recompilation item are examined. New recompilation items are created for the constants of the references according the modification type of the first item and the type of the reference. This process continues until all affected constants have been collected.

Function: `pc%who-refers-inside-this-scope`
Arguments:

`desc`
`scope`

Documentation: Given a descriptor `desc` returns all references to that constants inside `scope`.

Function: `pc%scope-visible-to-constant-p`
Arguments:

`constant`
`scope`

Documentation: Returns T if definitions of `scope` are visible in the definition of `constant`. Otherwise returns NIL.

Function: `pc%affected-generator-instances`

Arguments:

generators

Documentation: The instances of every (changed or affected) generator in generators are considered. If the modification type of the recompilation item of the generator is :deleted, :header-changed, or :module-type-changed then the instances must be deleted and therefore the pasts of those instances are inserted into set old-equates. Otherwise a new past is created for the instances by pc%resubst. The new pasts are inserted into new-equates and the old pasts are inserted into old-equates. Then pdiff$calculate-equate-changes is applied to old-equates and new-equates and the equate changes are propagated recursively with pc$propagate-changes. The created recompilation items are returned.

Function: pc%reference-types
Arguments:

ref

Documentation: Returns five values that describe the types of the reference: headerp, bodyp, bindp, inlinep, and isp (see the documentation of module refs).

Function: pc%resubst
Arguments:

instance

Documentation: Whn the generator of instance has changed this function is used in the creation of a new past for the instance.

Function: pc$clear-items
Arguments:

items

Documentation: A cleanup routine that is used when an internal error breaks the computation of changed or affected constants.

Function: pc$recompile
Arguments:

items
deleted-scopes

Documentation: This function processes the recompilation items in items according the modification types. If the modification type of an item is :equates-changed the past of the constant of the item is changed, the scopes are copied, and the source file name is updated. If the modification type is :deleted the constant of the item is deleted from the symbol table. If the modification type is :code-affected the status of the constant of the item is changed to needs-codegeneration, i.e., the constant is inserted into the code generation queue. Otherwise the constant is inserted into the recompilation queue by changing its status to needs-checking and the constant is reinitialized. Finally the function deletes deleted-scopes.

Function: pc%copy1-scopes
Arguments:

old-past
new-past

Documentation: Given two module equates `old-past` and `new-past` copies the top level scope of `old-past` to `new-past`.

Macro: `pc%copy-scope-cond`
Arguments:
  `old-expr-var`
  `new-expr-var`
Documentation: A shorthand that is used in the implementation of `pc%copy-scopes`.

Function: `pc%copy-scopes`
Arguments:
  `old-past`
  `new-past`
Documentation: Copies the scopes of equate `old-past` to the equate `new-past`. Does not copy recursively the scopes of the equates in the beginning of the module of `old-past`.

17 The Context Package

The context package contains modules *symbol-table*, *context*, and *scope-print*. The *scope-print* is an auxiliary facility used for debugging purposed only.

17.1 The symbol-table

The module *symbol-table* uses *hash-table* to create a symbol table. The key used is a pair of string and integer. The string is the name of the symbol and the integer is the id of the scope of the symbol. Keys are equal if both the string and the scope id are same.

Functions *#'string&int-=*, *#'string&int-hash-key*, and *#'string&int-assoc* are defined to be used as *:key-datum-=*, *:hash-key-function*, and *:chain-assoc-function* for the hash-table. The initial size of the table is 2000.

17.2 The Context

The *context* module contains the definitions of **scope**, **scope pool**, **context**, **routine-type pool**, and **signal pool**.

17.2.1 The scope datatype

The scopes are defined using the descriptor language. They all have fields: *id, father, children, persistent-p, unordered, splice-child, pending-marker, dont-delete-p,* and *def-list*. The descriptors are of types *scope, redefine-scope, stop-search-scope, root-scope*. They form a linear hierarchy so that the *scope* is only of that type and the *root-scope* of all types. *redefine-scope* is a scope which allows several definitions for a name, *stop-search-scope* is a scope beyond which the search is not continued in the tree, and *root-scope* is the type of the root scope of the tree.

The fields are: *id*, an integer unique to that scope, an optional *father*, a list of *children*, a boolean *persistent-p*, which is true when the scope cannot be deleted, a boolean *unordered* which is true, when the definitions in the scope are unordered, *splice-child* which contains the reference to the next part of the spliced scope, a *pending-marker* which is a reference to a function which is to refer to this scope if the scope becomes persistent, a boolean *dont-delete-p*, which is true if the scope is not to be deleted by the recompilation, and *def-list* which contains the list of definitions in the scope.

The operations are:

Function: `scope$mark-persistent`
Arguments:

```
scope
```

Documentation: Marks the scope *scope* persistent. If pending marker is set, funcalls it. The persistensy is recursive, parents of a persistent scope are persistent.

Function: `scope$link-and-set-attributes`
Arguments:

```
p-scope
redefine-p
unordered
&optional (stop-search t)
```

Documentation: Creates a scope and sets its attribute.

Function: `scope$release-definitions`
Arguments:

```
scope
```

Documentation: Removes all definitions from the `scope`.

Function: `scope$insert`
Arguments:

```
scope
name
object
```

Documentation: Inserts the `object` into `scope` with name `name`.

Function: `scope$merge`
Arguments:

```
scope1
scope2
```

Documentation: Merges scopes `scope1` and `scope2`.

Function: `scope$find-in-this-scope`
Arguments:

```
scope
name
&optional return-even-undef nil
```

Documentation: Finds the object associated with `name` from `scope`. If `return-even-undef` is set, returns even undefined descriptors. Doesn't traverse the scope tree.

Function: `scope$find`
Arguments:

```
scope
name
&optional pass-redefine-p t
          set-import-allowed nil
          return-even-undef nil
```

Documentation: Finds the object associated with `name` from `scope`. If `pass-redefine-p` is set, search passes redefine-scopes borders. If `set-import-allowed` is set imports the object from included generations if possible. If `return-even-undef` is set, returns even undefined descriptors.

Function: `scope$remove`
Arguments:

```
scope
name
&optional pass-redefine-p nil
```

Documentation: Removes the object associated to the `name` from the `scope`. Is recursive.

Function: `scope$search-root`
Arguments:

```
scope
```

Documentation: Recursively searches the root scope from `scope`.

Function: `scope$depth-first`
Arguments:

```
root
function)
```

Documentation: Applies the `function` to the subtree below `scope` in a depth-first manner.

Function: `scope$new-child-scope`
Arguments:

```
parent-scope
```

Documentation: Creates a new child for `parent-scope`.

17.2.2 The scope-pool

The scope-pool is meant to be used as a pool of scopes. It is organized according to the type of the scopes into four lists. Currently the scopes are not reused, i.e., they are not returned into the pool when they have been released. This is due to the difficulty of knowing when a scope is totally unreachable. So the lists in the scope pool are empty and requests always create new scopes.

The operations are:

sp$new		$\longrightarrow$ scope-pool
sp$root-scope	scope-pool	$\longrightarrow$ root-scope
sp$stop-search-scope	scope-pool	$\longrightarrow$ redefine-scope
sp$redefine-scope	scope-pool	$\longrightarrow$ sp$scope
sp$release-scope	scope-pool, scope	$\longrightarrow$

17.2.3 The context datatype

The context contains the state of the compiler. The four name spaces and a whole lot of other status information. Its representation is a struct having fields:

in-header-p	A boolean flag. T if compiling the header of a module, `nil` otherwise.
name-scope	The current name scope, i.e., a pointer to the name scope tree.
handler-scope	The current handler scope.
signal-scope	The current signal scope.
iteration-scope	The current iteration scope.
statement-stack	A stack of containing statements, used ,e.g., to find a containing loop statement.
locals	A list of local variables found so far in the module under compilation.
module-definition-name-scope	The name scope, where the module under compilation is defined.
referee	The constant corresponding to this context.
module	The module corresponding to this context.
supermodule	The supermodule of the `module` or `nil`.
datatype	The datatype the `module` belongs to or `nil`.
generator	The generator surrounding the `module`.
instance	The instance surrounding the `module` or `nil`.
instantiation-point	The point in source, where the instance was made.
uses	The assoc list of uses and the corresponding `pasts`.
first-name-scopes	A list of top level name scopes of the `past` of the constant corresponding to this module. A generator instantiated with different parameter values can have more that one first-name-scopes.
unhandled-signals	A list of unhadled signals.

The operations: **Function:** `context$new`
Arguments:

```
name-scope
&optional supermodule
          datatype
          generator
          instance
          instantiation-point
          uses
```

Documentation: Creates a new context using argument values and returns it.

Function: `context$initialize`

Arguments:

    context
    name-scope
    supermodule
    datatype
    generator
    instance
    instantiation-point
    uses
    first-name-scopes

Documentation: Initializes the fields of the `context` to the values given.

Function: `context$pseudo-p`
Arguments:

    context

Documentation: Returns T if the module corresponding to the context is within an pseudo instance.

Function: `context$print`
Arguments:

    context
    stream
    name-space

Documentation: Prints the scopes of the given `name-space` of the `context` to the `stream`.

Function: `context$current-name-scope`
Arguments:

    context

Documentation: Returns the current name scope of `context`.

Function: `context$find-loop`
Arguments:

    context

Documentation: Returns the nearest loop statement from the statement stack if any.

Function: `context$find-iterates`
Arguments:

    context
    handler

Documentation: Returns the list of iterate statements below `handler` from the statement stack.

Function: `context$find-bind`
Arguments:

    context
    name

Documentation: Finds the definition associated with `name` from the scope containing the module associated with the `context`.

Function: `context$find`
Arguments:

```
context
name
name-space
&optional even-undef nil
```

Documentation: Finds the object associated with `name` in `name-space` (:name, :handler, :signal, :iteration) from `context`. If `even-undef` is set returns even undefined descriptors.

Function: `context$find-and-make-if-not`
Arguments:

```
context
name
name-space
&optional context-for-undefined *context*
```

Documentation: Finds an object named `name` from `name-space`. If it does not exist creates an undefined constant into `context-for-undefined` and returns it.

Function: `context$find-in-this-name-scope-and-make-if-not`
Arguments:

```
context
name
```

Documentation: Finds an object named `name` from name name-space without traversing the scope tree. If it does not exist creates an undefined constant into this name scope and returns it.

Function: `context$insert`
Arguments:

```
context
object
&optional splice-allowed t
```

Documentation: Inserts the `object` into `context`. Chooses the name space according to the type of the object. If `splice-allowed` is T allows an insert into persistent scope by splicing.

Function: `context$insert-statement`
Arguments:

```
context
statement
```

Documentation: Pushes the `statement` on top of the statement stack into `context`.

Function: `context$remove`

Arguments:

```
context
name
name-space
```

Documentation: Removes the object named `name` in `name-space`.

Function: `context$remove-statement`
Arguments:

```
context
```

Documentation: Removes the topmost statement from the statement stack. (Removes all the when handlers on top of it, too).

Function: `context$open-scope`
Arguments:

```
context
name-space
&optional unordered nil
          redefine-p nil
```

Documentation: Opens a new scope into `name-space`. Uses `unordered` and `redefine-p` as attributes for the new scope.

Function: `context$close-scope`
Arguments:

```
context
name-space
&optional even-persistent nil
```

Documentation: Closes a scope from `name-space`. If `even-persistent` is T closes even the persistent scope.

In addition to the operations there exist three special forms:

```
(defmacro context$in-new-scope
  ((context name-space
    &optional (unordered {\tt nil}) (redefine-p {\tt nil}))
    &body body)
```

Opens a scope into `name-space`, executes the `body`, and closes the scope. Uses `unordered` and `redefine-p` as arguments for the new scope.

```
(defmacro context$in-name-scope ((context scope) &body body)
```

Executes the `body` in name scope `scope`. After execution sets the name-scope of `context` to its original value.

```
(defmacro context$within-statement ((context statement) &body body)
```

Pushes the `statement` into the statement stack, executes the `body`, and pops the statement stack.

17.2.4 The routine type pool

The routine type pool contains references to all routine types. All types are created using it, so all equal routine types are eq. Types are organized into a tree according to the argument types, signals and return/yield types.

The trees are created using the macro: **pr$routine-type**. The types are referred to using the function: **pr$proc-like**.

17.2.5 The signal pool

Signals are stored in ***context$signal-store-scope*** according to their name. The signals with different types are organized as a list in the struct **signal-item**.

18 The Data Base Package

The data base package contains modules *generation-database*, *module-database*, and *module-database-browser*. The generation-database implements the generation data base, the module-database implements the module data base, and the module-database-browser implements a browser for it.

18.1 The Generation Data Base

The generation data base is used to store the generations and systems. Information is kept in the file system, one file per each system and generation. The system files are kept in a system-base directory and the generations in directories generated for each system. The system and the generation files are append files and display the development history of the corresponding object.

The `world`, a struct containing the system and generation counters is stored using function **world$save** and retrieved using function **world$load**. The world is stored into `-world-save-file-`.

The systems are stored in `-system-directory-`. The objects stored into the system file are:*name, id, generation-counter, root-generation-id, newest-generation-id.* The objects are stored as a pair of a reload operation name and an object. The reload operation are associated to the type in constant `-system-db$operation-alist-`.

The top-level operations are:

Function: `system-db$load`
Arguments:
    `system-name`
    `&key force-loading nil`
Documentation: Loads a system named `system-name` and returns it. If `force-loading` is T, the system is reloaded, even if it was loaded already.

Function: `system-db$exists-p`
Arguments:
    `system-name`
Documentation: Returns T if system name `system-name` exists in data base.

Function: `system-db$delete`
Arguments:
    `system-name`
Documentation: Deletes a system named `system-name` from the data base.

Function: `system-db$create`

Arguments:

`system`

Documentation: Creates a data base entry for `system`.

Function: `system-db$write-object`

Arguments:

`system`
`object`
`operation`

Documentation: Writes the `object` of type `operation` into the file describing the `system`.

The file describing a generation is organized basically in the same way as the file describing a system. However the operations loading the objects from the history file are more complex and have been written one by one instead of using a macro to create them. There are also more object types.

The loading order is such that the files in the generation are collected into a hash table overwriting the previous definitions if they exist, and the export commands are collected into a list. After all modules have been loaded and the generation compiled, the modules in the export list are exported.

The object types are: *id, local-id, system, include, included-by, father-generation, child-generation, insert-module, delete-module, export-module.* The operations associated to them are listed in `-generation-db$operation-alist-`

The top level operations are:

Function: `generation-db$write-object`

Arguments:

`generation`
`object`
`operation`

Documentation: Writes the `object` and the load operation of its type `operation` into the file describing `generation`.

Function: `generation-db$create`

Arguments:

`generation`

Documentation: Creates a description file for `generation`.

Function: `generation-db$delete`

Arguments:

`generation`

Documentation: Deletes the `generation` from the data base.

Function: `generation-db$load-top`

Arguments:

  `system-name`
  `local-id`
  `&key operation :load`

Documentation: Loads and returns the generation of `system-name` with id `local-id`. The `operations` are `:load, :create-child, and :create-new`.

Function: `generation-db$exists-p`
Arguments:

  `system-name`
  `local-id`

Documentation: Returns T if generation `local-id` from system `system-name` exist.

Function: `generation-db$export-module`
Arguments:

  `generation`
  `name`

Documentation: Exports the module name `name`.

Function: `generation-db$delete-module`
Arguments:

  `generation`
  `name`

Documentation: Deletes the module name `name`.

18.2 Module data base

Information is kept in the file system, the in-core directory exist only to speed up access. Modifications are always made both to the file system and the directory. The in-core directory structure is a one-to-one copy of the file structure. The information in the in-core directory is encoded in the pathname of the module source.

The directory is divided according to the module type into hash tables. They contain the modules in a two dimensional array indexed by the version and revision numbers. The array is dynamic, so that space is not wasted.

Function: `mb$create`
Arguments:

Documentation: Creates an in-core module directory and sets it to be the value of `*mb$directory*`.

Function: `mb$insert`

Arguments:

`modulename`
`type`
`version`
`revision`
`source`

Documentation: Inserts a module named `modulename` of type `type` into the data base. `source` is a list of lines. If there already existed a module of the same type, name, version and revision, the revision counter is incremented.

18.3 The Module Base Browser

The module base browser is implemented using the `define-program-framework` of the lisp-machine. A presentation type is defined for each type displayed. The operations are defined with `define-presentation-action` and `define-presentation-to-command-translator`.

19 The Lisp Code Generator Package

The lisp code generator compiles the abstract syntax trees generated by the checker to lisp code. The compilation is defined using the generic function facility. Three lisp packages are defined for the symbols generated. They are *xe-fun, xe-var,* and *xe-loc*.

The constants which need codegeneration are collected into a list named **status$needs-codegeneration**. The list is ordered using function *cg$ordered-list-of-modules-that-need-codegeneration*.

The compilation process can be described as a transformation from cast specification to list code. The corresponding abstract syntax generic function is named *lcg%compile-node*. The global information needed to reserve variable names and labels is stored in a structure called *global-scope-structure*.

The top level codegeneration function is: *lcg$compile-descriptor*. It compiles the descriptor according to its type. Datatypes are compiled to their own variable declarations, foreign procedures to a function calling the external name, and routines to a lisp-function doing the necessary operation. Other types of descriptors cannot be compiled.

19.1 The transformation

COMPILE ROUTINE:
Input: `procedure or procedure-instance descriptor`
Output:

```
(defun <internal-name> (<arg-names>
                        &aux <local-names> <other-locals>)
  (block __rblock
    (tagbody
       [<own-initializations>]
       #source-cast#
       (go _end)
       _throw
       (throw 'signal ?signal)
       _end)))
```

COMPILE-ROUTINE:
Input: `an iterator or an iterator instance`
Output:

```
(defun <internal-name> (<arg-names>
                        &aux <local-names> <other-locals>)
```

```
  (setf ?fn #'(lambda (?path)
               (block __rblock
                 (tagbody
                   (case (pop ?path)
                     #goto-labels#)
                   #source-cast#
                   (go _end)
                   _throw
                   (throw 'signal ?signal)
                   _end))))
   (funcall ?fn nil))
```

STATEMENT-ASSIGN:

Input: `(statement (:assign expr &aux (vars desc-list)))`

Output:

```
    If there is more than one variable then:
            (multiple-value-setq (<var-names>) #expr#)
    else
            (setq <(car var-names)> #expr#)
```

STATEMENT-INVOCATIONS

Input:

```
        (statement (:invocations expr))
```

Output:

```
    #expr#
```

STATEMENT-IF:

Input:

```
         (statement (:if (cond-acts &list1 condition-act)
                          (else &opt statement))
         (condition-act (expr statement))
```

Output of statement-if:

```
        #cond-acts#
        #else#
        ?end-label
```

Output of condition-action:

```
        (if (zerop #expr#)
            (go ?labelN))
        #statement#
        (go ?end-label)
        ?labelN
```

STATEMENT-CASE:
Input:
```
(statement (:case expr
                  (case-options &list case-option)
                  (otherwise &opt statement))
(case-option ((case-list &list expr) statement))
```

Output of statement-case:
```
(setf <var-name> #expr#)
#case-options#
#statement#
?end-label
```

Output of case-option:
```
(if (not (or (eql <var-name> #(elem-from case-list)#))
    (go ?labelN))
#statement#
(go ?end-label)
?labelN
```

STATEMENT-WHILE:
Input:
```
(statement (:while expr (statement &opt)))
```

Output:
```
?begin-label
(if (zerop #expr#)
    (go ?end-label))
#statement#
(go ?begin-label)
?end-label
```

STATEMENT-FOR:
Input:
```
(statement (:for (invocation expr)
                 (statement &opt)
                 &aux (vars desc-list)))
```

Output:
```
(multiple-value-setq (?fn ?path <var-names>)
                     #invocation#)
?begin-label
```

```
(if (null ?fn)
    (go ?end-label))
#statement#
?next-label
(multiple-value-setq (?fn ?path <var-names>)
   (funcall ?fn ?path))
(go ?begin-label)
?end-label
```

STATEMENT-ITERATE:
Input:

```
(statement (:iterate (iterate-defs &list iteration)
                     (statement &opt)))
(iteration ((invocation expr) (iterate &opt statement)
            &aux (name name)
                 (vars desc-list)))
```

Output:

```
#iterate-defs#
#statement#
```

Output of iteration:

```
(multiple-value-setq (iteration-fn iteration-path yield-vars)
   #expr#)
```

STATEMENT-NEXT:
Input:

```
(:next (iterations &list iteration))
```

Output for each iteration:

```
(multiple-value-setq (fn path yield-vars)
  (funcall fn path))
```

STATEMENT-BREAK:
Input:

```
(statement (:break (loop statement))
```

Output:

```
(go <loop-end-label>)
```

STATEMENT-CONTINUE:
Input:

```
(statement (:continue (loop statement)))
```

Output:

```
(go <loop-next-field>)
```

STATEMENT-RETURN:
Input:

```
(statement (:return (values &list expr)))
```

Output:

```
(return-from __rblock (values #values#))
```

STATEMENT-YIELD:
Input:

```
(statement (:yield (values &list expr)))
```

Output:

```
(return-from __rblock
   (values ?fn <(append ?path ?resume-label)> #value#))
?resume-label
```

STATEMENT-SIGNAL:
Input:

```
(statement (:signal (values &list expr) &aux (signal desc)))
```

Output:

```
(setf ?signal (list <signal-name>
                    #values#))
(go _routine-throw)
```

STATEMENT-EXIT:
Input:

```
(statement (:exit (handler when-or-star-handler)
                  (iterates &list statement)))
```

Output:

```
(go <handler-label>)
```

STATEMENT-EXCEPT:
Input:

```
(statement (:except (statement &opt)
                    (when-handlers &list when-or-star-handler)
                    (others-handler &opt))
(when-handler ((statement &opt)
               &aux (signal-names name-list)
                    (locals desc-list))
(star-handler ((statement &opt)
               &aux (signal-names name-list)))
(when-or-star-handler (:when-handler when-handler)
                      (:star-handler star-handler)))
(others-handler ((statement &opt)
                 &aux (var desc)))
```

Output of statement-except:

```
?begin-label
(multiple-value-setq (?signal-name ?signal-id ?values)
        (catch 'signal
           (tagbody
             (case (pop ?path) #goto-labels#)
             #statement#
             (go ?end-label))))
#when-handlers#
#other-handler# or
(throw 'signal (values ?signal-name ?signal-id ?values)))
?end-label
```

Output of when-handler:

If there is more than one exception name in handler and this is an explicit handler (i.e. (desc$starp handler) is NIL.

```
(if (not (member ?signal-id
                 <(desc$get-exception-names handler)>))
    (go ?next-label))
?label
#statement#
(go ?end-label)
?next-label
```

If there is only one exception name in handler and this is an explicit handler:

```
(if (not (eq ?signal-id
             <(car (desc$get-exception-names handler))>))
    (go ?next-label))
```

```
?label
#statement#
(go ?end-label)
?next-label
```

If there is more than one exception name in handler and this is an implicit handler (i.e. (desc$starp handler) is T).

```
(if (not (member ?signal-name
                 <(desc$get-exception-names handler)>))
    (go ?next-label))
?label
#statement#
(go ?end-label)
?next-label
```

If there is only one exception name in handler and this is an implicit handler:

```
(if (not (eq ?signal-name
             <(car (desc$get-exception-names handler))>))
    (go ?next-label))
?label
#statement#
(go ?end-label)
?next-label
```

STATEMENT-RESIGNAL:

Input:

```
(statement (:resignal (statement &opt)
            &aux (exceptions desc-list))
?begin-label
(multiple-value-setq (?signal-name ?values)
        (catch 'signal
           (tagbody
             (case (pop ?path) #goto-labels#)
             #statement#
             (go ?end-label))))
(if (not <test>)
    (go ?label))
(setf ?signal (list ?signal-name ?values))
(go _routine-throw)
?label
(throw 'signal (values ?signal-name ?values))
?end-label
```

STATEMENT-BEGIN-END:
Input:

```
(statement (:begin-end (statements &list statement)))
```

Output:

```
#statement1#
#statement2#
...
#statementN#
```

EXPR-LITERAL:
Input:

```
(expr (:literal literal)
```

Output:

```
<literal-value>
```

EXPR-CAST:
Input:

```
 (:cast expr)
```

Output:

```
#expr#
```

EXPR-VAR:
Input:

```
(expr (:var var))
```

Output:

```
<var-name>
```

EXPR-TYPED-OPERATION:
Input:

```
(expr (:typed-operation routine))
```

Output:

```
#'<routine-name>
```

EXPR-INVOCATION:
Input:

```
        (expr (:invocation expr (args &list expr))
```

Output:

```
If invocation is an expr-typed-operation:
        (<expr-typed-operation-routine-name> @#args#)
else
        (funcall #expr# @#args#)
```

EXPR-STRUCT-CONS:

Input:

```
        (expr (:struct-cons (fields &list1 struct-field)))
```

Output:

```
        (make-array <number-of-fields> :initial-contents #fields#)
```

EXPR-SEQUENCE-CONS:

Input:

```
        (expr (:sequence-cons (values &list expr)))
```

Output:

```
(make-array <number-of-values> :initial-contents #values#)
```

EXPR-CAND:

Input:

```
        (expr (:cand (expr1 expr) (expr2 expr)))
```

Output:

```
        (and #expr1# #expr2#)
```

EXPR-COR:

Input:

```
        (expr (:cor (expr1 expr) (expr2 expr)))
```

Output:

```
        (or #expr1# #expr2#)
```

20 The Intel Code Generator Package

The intel code generator and the optimizer are described in [7].

21 The ZMACS Package

The *zmacs* package consists of module *zmacs-init* and *xe-mode*. The zmacs-init contains functions which load the newxe system source files into zmacs buffers. This facility is aimed for system developers only.

The module *xe-mode* implements a major mode for the ZMACS editor called **xe-mode**. The usage of xe-mode is documented in [1]. The xe-mode is used when editing XE source code. XE source code is edited in XE buffer, which has xe-buffer-module-id, i.e., in buffer plist there is a property *:xe-module-name* and the value of that property is a list of generation, module name, version number, and revision number.

In xe-mode the ZMACS editor communicates quite a lot with the current XE compiler process. The communication is synchronous, i.e., the editor always waits for the results. In addition to the commands the xe-mode module containts a number of auxliary routines. They are docomented first:

21.1 XE Interval Line Stream

Xe interval line stream is represented by struct:

```
(cl:defstruct (xe-interval-line-stream (:conc-name xils$))
  last-line
  line)
```

It is an interval in the buffer from `line` to `last-line`. The operations are:

Function: `open-xe-interval-stream-with-bps`
Arguments:
    `first-bp`
    `last-bp`
Documentation: Creates an interval stream from `first-bp` to `last-bp` (Both buffer pointers.) and returns it.

Function: `xe-read-line-in-buffer`
Arguments:
    `stream`
    `eof-error-p`
    `eof-value`
Documentation: Returns the next line from interval stream `stream`. If `eof-error-p` is T signals an error, when attempting to read past last-line, otherwise returns `eof-value`.

21.2 Information retrieval

The following information retrieving routines are used all around the xe-mode.

Function: `user-name`
Arguments:

Documentation: Returns the name of the user currently logged in as a string.

Function: `get-xe-process`
Arguments:

Documentation: Returns the current xe-compiler-process. If there is none barfs an error.

Function: `get-current-generation`
Arguments:

Documentation: Returns the current generation, i.e., the value of `*system$current-generation*` in the current compiler process. If the value is `nil` barfs an error.

21.3 XE buffer handling

Function: `xe-new-version-or-revision`
Arguments:
`new-version-p`
Documentation: Creates a new version or revision (according to the `new-version-p`) of the definitions in the current buffer. Uses `split-stream` to divide the buffer according to the definitions and creates new XE buffers when neeeded.

xe-buffer-module-id Returns the xe-module-id.

set-xe-buffer-module-id Sets the xe-module-id.

xe-buffer-parser-error-msgs Returns the parser error message list.

set-xe-buffer-parser-error-msgs Sets the parser error message list.

xe-buffer-checker-error-msgs Returns the checker error message list.

set-xe-buffer-checker-error-msgs Sets the checker error message list.

make-xe-buffer-name Creates a xe buffer name string.

create-xe-buffer Creates a new xe buffer.

insert-text-into-xe-buffer Append the given pice of text to the buffer.

flush-xe-buffer Kill the text in the buffer.

find-xe-buffer Searches for the buffer corresponding to the given module.

delete-xe-buffers-of-other-generations Kills the buffers not belonging to the generation given as argument.

update-this-buffer-checker-error-list Updates the checker error message list of the buffer given as argument.

update-all-buffer-checker-error-lists Updates the checker error messages lists of all buffers.

21.4 Item window

The item window is used to edit the error messages. It is a generic facility for item editing. It split the buffer in two parts, the upper one showing the items, and the lower one the buffer they are associated to. It is represented by a struct:

```
(defstruct (item-window-data (:conc-name iwd$))
  item-line-buffer-name
  item-action
  item-list-func
  item-line-output-func
  item-action-func
  item-point-func
  item-sort-p
  item-cmp-func)
```

The items to be manipulated are created calling `item-list-func`. The action applied to them is `item-action`, the lines are displayed using `item-line-output-func`, `item-point-func` gives the point associated to the item, `item-sort-p` determines whether the items are to sorted or not and `item-cmp-func` is used to sort them.

The commands implemented using the item window are:

com-xe-edit-error-msgs Edits error messages associated with the current buffer.

com-xe-edit-next-item Edits the next item.

com-xe-edit-prev-item Edits the previous item.

com-xe-quit-editing-items Quits the edit messages mode.

21.5 Commands for Systems and Generations

com-xe-clear Clears all data structures of the compiler.

com-xe-create-system Creates a new XE system. Asks for system name and checks that it does not already exist. Uses functions `system-db` and `generation-db` functions to create the system and its first generation and sets them to be the current system and generation.

com-xe-edit-generation Asks for a generation, loads it, and sets it to be the current generation. Loading is done in the XE process.

com-xe-include-generation Asks for the generation to be included, loads it in the XE process, and includes it into the current generation. Given an numeric argument asks for another generation and uses it instead of the current generation.

21.6 Commands for Modules

com-xe-edit-module Given a numeric argument asks for a generation, otherwise uses the current generation. Asks for a module name. If the module is already loaded to some buffer selects it, otherwise checks if the generation has a module of the given name. If it does not have, asks a confirmation to create the module and a buffer to edit it, otherwise creates a new XE buffer and loads the source code of that module into it.

com-export-module Given a numberic argument asks for a generation otherwise uses the current generation. Asks for a module name, and checks if the generation has a module with that name. If it has, exports it otherwise barfs an error.

com-export-all Given a numeric argument asks for a generation otherwise uses the current generation. Then exports all the modules of that generation.

com-xe-check-module Given a numeric argument asks for a generation otherwise uses the current generation. Asks for a module name and checks if it exists. If it does, sends it to the checker.

com-xe-new-revision Creates a new revision of the definition in the buffer. Calls *xe-new-version-or-revision*.

com-xe-new-version Creates a new version of the definition in the buffer. Calls *xe-new-version-or-revision*.

com-xe-check-modified-definitions Sends a *check modified definitions* command to the current checker process.

com-xe-codegenerate Sends the *codegenerate* command to the current checker process.

22 The Process Package

The process package contains modules *process-aux* and *process*. *process-aux* contains the process variables and the *cp$in-xe-process* macro, which executes the *body* in the process given as argument, by sending the *body* to the checker within a lambda and waits for the results.

The *process* module contains *checker-process-top*, the commands that communicate with it, and a set of the most common operations. The routines in the beginning of the module implement the connection between the process and the window.

The *cp$checker-process-top* consists of a loop inside a loop. The checker never exits the outer loop. Around the inside loop there is a catch for *cp%continue*, and a let, which binds the global variables referred to inside the checker. This construction forms a foolproof method for checker initialization. The inside loop is an *error-restart* loop, in which the checker waits for a command. When the command arrives in to the command queue it is funcalled.

The functions, which communicate with the checker are: *cp$send-command-to-checker* and *cp$send-command-to-checker-and-wait*. They both insert the command to the tail of the checker queue. The latter waits for the checker to complete the operation, and returns the result of the evaluation of the command.

The rest of the functions implement some often used facilities. Their operation is straight forward and their names are self explanatory.

23 The User Interface Package

The user interface package contains only one module: frame. It defines a flavor *xe-ui-frame* of type *tv:bordered-constraint-frame* 5. It has four panes: *editor*, *header*, *compiler*, and *error-messages*. The frame is selectable by select key “u”, the frame inits the compiler process in an after init method. The code is very straight forward.

Figure 5: The frame

input lib.tex

References

[1] XE Programming Environment Users's Guide. Juha Kuusela, Jari Arkko, Vesa Hirvisalo, Esko Nuutila, Markku Tamminen, Helsinki Univ. of Tech., Dept. of Comp. Sci. report TKO-C35, 1989.

[2] XE Reference Manual. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, Markku Tamminen, Helsinki Univ. of Tech., Dept. of Comp. Sci. report TKO-A26, 1989.

[3] CLU Reference Manual. B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. Schaffert, R. Scheifler, A. Snyder. Springer-Verlag, Berlin, 1981.

[4] AST and DG Tools Programmer's Guide. Esko Nuutila, Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Markku Tamminen, Helsinki Univ. of Tech., Dept. of Comp. Sci. report TKO-C39, 1989.

[5] A Practical Toolkit for Making Portable Compilers, A S Tanenbaum et al., CACM, vol. 26, no. 9, pp 654-660, September 1983.

[6] Common LISP: The Language. Guy L. Steele Jr. Digital Press, 1984.

[7] Asiantuntijajärjestelmäkielen optimoitu toteutus, Jari Arkko, Diplom thesis, Helsinki Univ. of Tech., Dept. of Comp. Sci., 1989.

[8] ROCC Programmer's Guide. Jari Arkko, Vesa Hirvisalo, Juha Kuusela, Esko Nuutila, Markku Tamminen, Helsinki Univ. of Tech., Dept. of Comp. Sci. report TKO-C38, 1989.

Contents