Application Programming Interface (API)

The **A**pplication **P**rogramming **I**nterface (API) of SLADE consists of three modules defining constants, types, and actions. The API modules are:

- *evalattr* defines actions for the evaluation of attributes of the standard types *boolean*, *char*, *integer*, *real* and *string*.
- *idscope* defines types, constants and actions for maintaining information of identifiers and/or expressions.
- *vimcode* defines the instruction set of the **VI**rtual **M**achine (VIM) code interpreter, and actions for generating these (optionally labelled) instructions.

The following layout is used for describing the items of the API modules:

`Name` names the items,

`Definition` defines the named items, indicating how to use them. Reserved words are printed in **boldface**,

`Description` describes the functionality of the named items. The names of the items are printed in *italics*, and

`See also` refers to related data types and operations. Reserved words are printed in **boldface**.

Module *evalattr*

The API module *evalattr* defines actions for assigning, reading or writing attributes, which have the basic types *boolean*, *char*, *integer*, *real* and *string*. The actions with exactly one output parameter may also be used as default actions for synthesised attributes. The arguments of the write actions are written to the standard output.
In the remainder of this section the actions of the API module *evalattr* are named, defined, and described.

Boolean type actions

`Name` assign_bool, clear_bool, read_bool, write_bool.

`Definition`

```
actions
    assign_bool (out dest: boolean;  in src: boolean).
    clear_bool  (out dest: boolean).
    read_bool   (out value: boolean; in prompt: string).
    write_bool  (in value: boolean).
```

`Description` *assign_bool* assigns the boolean value *src* to *dest*.
clear_bool assigns the boolean value *false* to *dest*.
The boolean variable `some_var` can be set to *true* by calling `assign_bool(some_var, true)`.
read_bool prints the string *prompt* on standard output, reads a boolean value from standard input and stores the result in *value*.
write_bool writes the boolean value *value* to standard output.

See also **actions**
dy_op_bool(), mon_op_bool(); (Module vimcode)

Char type actions

Name assign_char, clear_char, read_char, write_char.

Definition
```
actions
    assign_char (out dest: char;     in src: char).
    clear_char  (out dest: char).
    read_char   (out value: char;    in prompt: string).
    write_char  (in value: char).
```

Description *assign_char* assigns the char value *src* to *dest.*
clear_char assigns the null-char value to *dest.*
read_char prints the string *prompt* on standard output, reads a character value from standard input and stores the result in *value*.
write_char writes the char value *value* to standard output.

See also **actions**
dy_op_char(), mon_op_char(); (Module vimcode)

Integer type actions

Name assign_int, clear_int, read_int, write_int.

Definition
```
actions
    assign_int  (out dest: integer;  in src: integer).
    clear_int   (out dest: integer).
    read_int    (out value: integer; in prompt: string).
    write_int   (in value: integer).
```

Description *assign_int* assigns the value *src* to *dest.*
clear_int assigns the value 0 to *dest.*
read_int prints the string *prompt* on standard output, reads an integer value from standard input and stores the result in *value*.
write_int writes the integer value *value* to standard output.

See also **actions**
dy_op_int(), mon_op_int(); (Module vimcode)

Real type actions

Name assign_real, clear_real, read_real, write_real.

Definition
```
actions
    assign_real (out dest: real;     in src: real).
    clear_real  (out dest: real).
    read_real   (out value: real;    in prompt: string).
    write_real  (in value: real).
```

Description *assign_real* assigns the real value src to dest.
clear_real assigns the real value 0.0 to *dest*.
read_real prints the string *prompt* on standard output, reads a real value from standard input and stores the result in *value.*
write_real writes the real value *value* to standard output.

See also **actions**
dy_op_real(), mon_op_real(); (Module vimcode)

String type actions

Name assign_string, clear_string, read_string, write_string.

Definition
```
actions
    assign_string(out dest: string; in src: string).
    clear_string (out dest: string).
    read_string  (out value: string;in prompt: string).
    write_string (in value: string).
```

Description assign_string assigns the string value src to dest.
clear_string assigns the empty string value to *dest*.
read_string prints the string *prompt* on standard output, reads a string from standard input and stores the result in *value.*
write_string writes the string value *value* to standard output.

Module *idscope*

The API module *idscope* defines types, constants and actions for maintaining information of identifiers and/or expressions. The information of all identifiers is stored in a binary tree, the *identifier tree*. The identifier information is found directly via the identifier's name. Each node of the identifier tree contains the information of one identifier being:

- the identifier name, and
- the *declaration list*.

The *declaration list* of an identifier is a push-down list of declaration information. The top of the declaration list contains the identifier's declaration information for the current (innermost) scope. Declaration information of the surrounding scope(s) is stored below the top of the declaration list.

The declaration information of an identifier with a particular scope is stored in a *definition record*. Definition records have to be created during the parsing of the *defining occurrence* (i.e. the occurrence in a declaration) of the identifier. Definition records have to be deleted if the scope in which the declaration was made is left. Some relevant information stored in the definition record is listed below (see also Figure 11):

- the name of the identifier (which is set by *insert_definition*),
- the data-segment number (which is set to the current scope level by *create_definition()*),
- dva (**d**isplacement (of a variable), **v**alue (of a single constant) or **a**ddress (of a procedure) set by *create_definition()*),
- type (set by *create_definition()* and *put_def_type()*),
- kind (set by *create_definition()* and *put_def_kind()*),
- access (set by *create_definition()* and *put_def_access()*),

- dimension (set by *create_definition()* and *put_def_dim()*, used in dynamic and static array definitions),
- list of bound pairs (used in static array and record definitions, set by *add_def_bound()*),
- number of child definitions (e.g. number of fields/parameters in record/procedure definitions, set by *add_def_child()*).
- pointer to child definition list (e.g. field/parameter list of record/procedure definitions, set by *add_def_child()*).
- pointer to next (child) definition (e.g. next field/parameter definition in record/procedure definitions, set by *add_def_child()*).

For each scope level a *scope record* is maintained. The scope record refers to a definition list, which is a list of valid identifier definition records within a particular scope.
The scope records are stored in a push-down list, the *scope list*, with the current scope record on top.

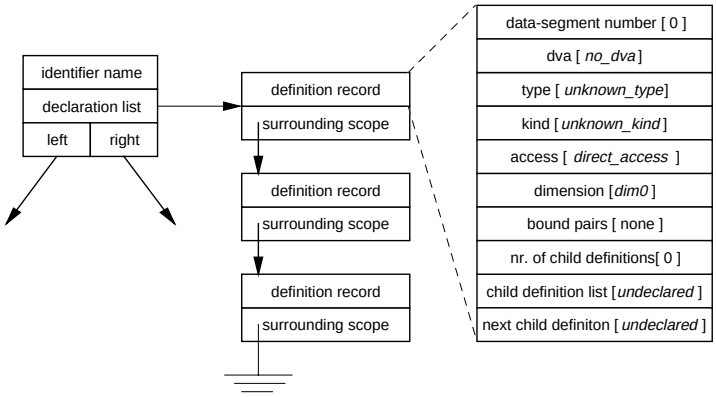

Figure 11: *The declaration list of definition record(s) inserted in a node of the identifier tree, with default values between square brackets.*

In the following example, identifiers are declared at different scope levels:

```
BEGIN a,b,c
   BEGIN d,e
      BEGIN a,g
```

At this point in the program text, the scope list contains two definition records of identifier *a*: one at level 3 and one at level 1. In Figure 12 the corresponding scope list is shown.
Note: Definition records can also be used as “attribute containers” of expressions; in that case they are not inserted in the identifier tree or scope list but merely used to pass on multiple relevant attributes (like type, kind, access, etc.) between production rules.

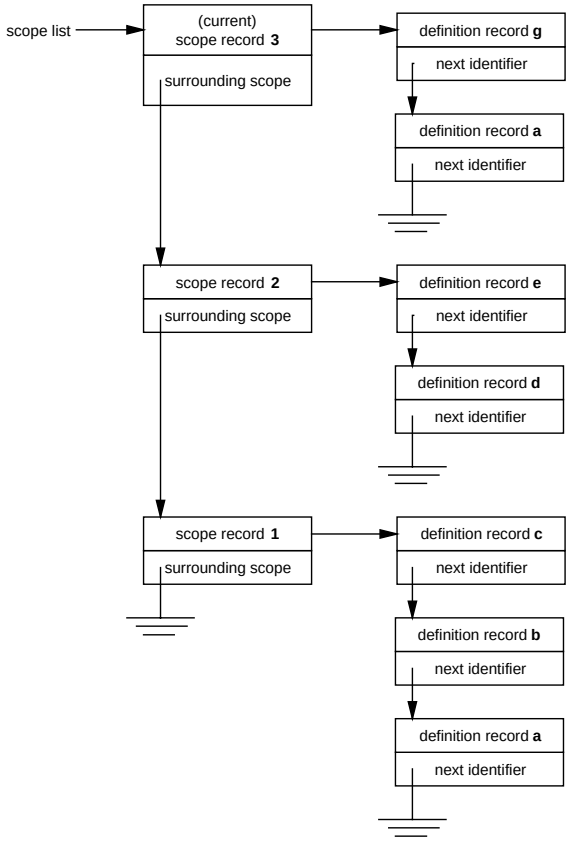

Figure 12: *An example of a scope list, with a scope record for each scope.*

Types and constants

Name def_ptr, e_access, e_kind, e_type, dim0, newline_char, no_dva, null_def, undeclared.

Definition

```
types
    def_ptr.
    e_access = (
       address_access, direct_access, indirect_access,
       readonly_access, unknown_access
    ).

    e_kind = (
       avar_kind, const_kind, proc_kind, ptr_kind,
       rec_kind, rpar_kind, svar_kind, tdecl_kind,
       unknown_kind, vpar_kind
    ).
    e_type = (
       bool_type, char_type, int_type, no_type,
       string_type, unknown_type
    ).
constants
```

```
dim0           : integer.
newline_char   : char.
no_dva         : integer.
null_def       : def_ptr.
undeclared     : def_ptr.
```

Description *def_ptr* is a pointer type pointing to a definition record.

e_access enumerates the available kinds of access:

- *address_access* to address variables and reference arguments,
- *direct_access* to access a variable directly by its segment number and displacement,
- *indirect_access* to de-reference pointer variables and reference parameters,
- *readonly_access* read-only access (can exist in combination with another access value),
- *unknown_access* unknown access.

e_kind enumerates the available kinds:

- *avar_kind* array variable,
- *const_kind* constant,
- *proc_kind* procedure,
- *ptr_kind* pointer,
- *rec_kind* record,
- *rpar_kind* reference parameter,
- *svar_kind* single variable,
- *tdecl_kind* type declaration,
- *unknown_kind* unknown kind,
- v*par_kind* value parameter.

e_type enumerates the available types:

- *bool_type* type boolean,
- *char_type* type character,
- *int_type* type integer,
- *no_type* no type,
- *string_type* type string,
- *unknown_type* unknown type.

dim0 is an integer constant to be used in definition records of 0-dimensional identifiers/expressions.
newline_char is a character constant defining the ASCII newline character.
no_dva is an integer constant to be used in definition records without a relevant **d**isplacement (of a variable), **v**alue (of a single constant) or **a**ddress (of a procedure)..

null_def is a pointer constant (of type *def_ptr*) pointing to a definition record describing the null pointer.
undeclared is a pointer constant (of type *def_ptr*) pointing to a definition record with all fields set to their default value.

See also

```
actions
    chk_def_access(), chk_def_dim(), chk_def_kind(),
    chk_def_type(), get_def_access(), get_def_dim(),
    get_def_kind(), get_def_type(), put_def_access(),
    put_def_dim(), put_def_kind(), put_def_type();
                                                (Module idscope)
    emit_load(), emit_store()                   (Module vimcode)
```

Scope administration actions

Name current_scope, enter_scope, exit_scope, list_scope.

Definition

```
actions
    current_scope  (out level: integer).
    enter_scope    ().
    exit_scope     ().
    list_scope     ().
```

Description

current_scope returns in parameter *level* the scope level of the program text currently being parsed.
enter_scope creates a new *scope record* on top of the *scope list* and initialises the corresponding definition list.
exit_scope deletes the definition records of all identifiers present in the definition list of the current scope record. The current scope record itself is also deleted and the scope list is pointed to the scope record of the surrounding scope, which now becomes the current scope record. If the action *exit_scope* is called without a preceding *enter_scope()*, the error message: **No current scope, exit_scope() failed** is reported.
If definition records are not used (that is they are not used as an argument of *emit_load()*, *emit_call()* or *use_definition()*), the warning: **Identifier <name> defined, but never set or used** is reported.
list_scope reports a brief summary of the contents of all valid definition records at the current scope level on the standard output.
The option *test parser* should be passed through to the built compiler; if not, this action has no effect.

See also

```
actions
    emit_load(), emit_call();                   (Module vimcode)
    use_definition();                           (Module idscope)
```

Definition record actions

Name add_def_bound, add_def_child, create_definition, delete_definition, find_definition, find_field, if_definition, insert_definition, list_definition, of_type, use_definition.

Definition actions

```
add_def_bound     ( in def: def_ptr;
                    in low: integer; in up: integer).
add_def_child     ( in parent: def_ptr;
                    in child: def_ptr).
create_definition ( out def: def_ptr;
                    in dva: integer; in t: e_type;
                    in k: e_kind; in dim: integer).
delete_definition ( in def: def_ptr).
find_definition   ( out def: def_ptr;
                    in idname: string).
find_field        ( out def: def_ptr;
                    in idname: string;
                    in recdef: def_ptr).
if_definition     ( out result: def_ptr;
                    in then_def: def_ptr;
                    in else_def: def_ptr).
insert_definition ( in idname: string;
                    in def: def_ptr).
list_definition   ( in def: def_ptr).
of_type           ( out def: def_ptr;
                    in typdef: def_ptr;
                    in dpl: integer).
use_definition    ( in def: def_ptr).
```

Description *add_def_bound* adds a bound pair *low*, *up* to the bound pair list of the definition record pointed to by *def*, the dimension field of *def* is incremented also. If low is greater then up the error message: **Lower bound *low* exceeds upper bound *up*** is reported.
This action is to be called when parsing static array declarations (dynamic arrays do not have any bound pairs because the bounds of dynamic arrays are not known at compile-time).

add_def_child adds a definition record *child*, created with *create_definition()*, to the end of the child definition list of definition record *parent*.
This action is to be called when parsing record/procedure declarations to add field/parameter definition records to the corresponding record/procedure definition record.

create_definition creates a new definition record pointed to by the output parameter *def*. The data segment number field of *def* is set to the current scope level. The parameters *dva*, *t*, *k*, and *dim* are assigned to the corresponding fields of *def*. If *k* equals *const_kind*, *proc_kind* or *tdecl_kind* the access field of *def* is set to *readonly_access*, and if *k* equals *rpar_kind* the access field of *def* is set to *indirect_access*. The remaining definition fields are set to their default values (see also Figure 11).

In Table 2 and Table 3 an overview is given how to fill the definition record for the defined and used occurrences of different kinds of identifiers.

	access	dimension	kind
array	direct_access	> 0	avar_kind
array type	readonly_access	> 0	tdecl_kind
constant	readonly_access	dim0	const_kind
constant array	readonly_access	> 0	avar_kind
constant record	readonly_access	1	rec_kind
field variable	direct_access	dim0	rec_kind
pointer	direct_access	dim0	ptr_kind
procedure	readonly_access	dim0	proc_kind
record	direct_access	1	rec_kind
record type	readonly_access	1	tdecl_kind
reference parameter	indirect_access	dim0	rpar_kind
single variable	direct_access	dim0	svar_kind
value parameter	direct_access	dim0	vpar_kind

Table 2: *Defined occurrence table.*

	access	dimension	kind
address of variable	address_access	dim0	svar_kind
array variable	direct_access	> 0	avar_kind
contents of address	indirect_access	dim0	ptr_kind or rpar_kind
field variable	direct_access	dim0	rec_kind
indexed variable	indirect_access or direct_access	dim0	avar_kind
procedure call	readonly_access	dim0	proc_kind
record variable	direct_access	> 0	rec_kind
reference parameter used as argument	address_access	dim0	rpar_kind
reference parameter used in expression	indirect_access	dim0	rpar_kind
single variable	direct_access	dim0	svar_kind

Table 3: *Used occurrence table.*

In Table 2 the following assumptions are made:

- Before declaring an identifier of kind array or record, a type definition record should be created in which the array or record type is defined.

- The moment the array or record identifier is declared, the corresponding type definition record can be retrieved using *find_definition()*. The found type definition record should then be passed through as the second argument of *of_type()*.
- Records should be defined as 1-dimensional identifiers having one bound pair with lower bound 0 and upper bound equal to the number of fields (see *add_def_bound()*).

delete_definition deletes a definition record pointed to by *def*, which was previously created by *create_definition*. Note that *def* will point to an undefined value afterwards.
Definition records which are already inserted in the identifier tree (by *insert_definition()*), cannot be deleted by calling *delete_definition()* (this is done by *exit_scope()*). Procedure parameter definition records can only be deleted by deleting the procedure definition record.

find_definition searches the identifier tree for the name *idname* and returns a copy of the found definition record pointed to by *def*. If the definition record of identifier *idname* is not found, the error **Identifier <name> not found** is reported and *def* is assigned the value *undeclared*.

find_field searches the child definition list of the definition record pointed to by *recdef*. If a field with name *idname* is found, then *def* will be pointed to a copy of the found definition record. If no field is found the error **No such field in this record** is reported and *def* is assigned the value *undeclared*.
Note: *recdef* should point to a definition record defining a record variable, not a record type.

if_definition is to be used to derive the definition record of a conditional if-statement from the definition record(s) of the then-part and the (optional) else-part of that if-statement.
If the *type* fields of the definition records pointed to by *then_def* and *else_def* differ, the *type* of *then_def* will be set to *no_type*.
If the *dimension* fields of the definition records pointed to by *then_def* and *else_def* differ, the *dimension* of *then_def* will be set to *dim0*.
At last the parameter *then_def* is assigned to the output parameter *result* and the parameter *else_def* is deleted by calling *delete_definition()*.

insert_definition inserts a definition record pointed to by *def* in the identifier tree. *def* is also inserted in front of the definition list of the current *scope record* (on top of the *scope list*).
If a definition record already exists for the identifier *idname* within the current scope, the error **Double defined identifier: <name>** is reported.
If there is no current scope (i.e. *enter_scope()* has not been called), the error **Insertion of <name> failed, no current scope** is reported.
If the insertion was successful the name field of the definition record pointed to by *def* will be set to *idname*.
By calling *exit_scope()* the corresponding definition records will be deleted.

list_definition reports a brief summary of the contents of the definition record pointed to by *def* on standard output.
The option *test parser* should be passed through to the built compiler, if not this action has no effect.

of_type is to be used when declaring identifiers of a structured (i.e. array or record) type, *of_type* will copy the definition record pointed to by *typdef* into the definition record pointed to by *def*. Then the data segment number of *def* will be set to the current scope level; the displacement of *def* will be set to *dpl* (the displacement of the child definitions of *def* are derived recursively), and the kind field of *def* will be set to *rec_kind* if the type field of *typdef* is *no_type* else it will be set to *avar_kind*.

use_definition marks the definition record pointed to by *def* as "being used". By doing so *exit_scope()* can be prevented from complaining about defined but unused identifiers.

See also

```
types
    def_ptr, e_access, e_kind, e_type    (Module idscope)
actions
    emit_bound(), emit_descr();          (Module vimcode)
    enter_scope(), exit_scope();         (Module idscope)
```

Get definition information actions

Name get_def_access, get_def_dim, get_def_kind, get_def_type.

Definition

```
actions
    get_def_access (out a: e_access; in def: def_ptr).
    get_def_dim    (out d: integer;  in def: def_ptr).
    get_def_kind   (out k: e_kind;   in def: def_ptr).
    get_def_type   (out t: e_type;   in def: def_ptr).
```

Description These actions get (retrieve) the field values of the definition record pointed to by *def*.
get_def_access returns in *a* the value of the *access* field of *def*.
get_def_dim returns in *d* the value of the *dimension* field of *def*.
get_def_kind returns in *k* the value of the *kind* field of *def*.
get_def_type returns in *t* the value of the *type* field of *def*.

See also **types**
def_ptr, e_access, e_kind, e_type (Module idscope)

Put definition information actions

Name put_def_access, put_def_dim, put_def_kind, put_def_type.

Definition **actions**

```
put_def_access (in a: e_access;  in def: def_ptr ).
put_def_dim    (in d: integer;   in def: def_ptr ).
put_def_kind   (in k: e_kind;    in def: def_ptr ).
put_def_type   (in t: e_type;    in def: def_ptr ).
```

Description These actions put (update) the field values of the definition record pointed to by *def*. If *def* equals *undeclared* or *null_def*, no fields are updated.
put_def_access copies *a* into the *access* field of *def*. The *readonly_access* value can exist in combination with other values of type *e_access*. So if *a* equals *readonly_access*, the (old) value of the *access* field of *def* remains intact and the read-only flag of *def* will be set. If *a* equals *unknown_access*, the read-only flag of *def* is cleared.
put_def_dim copies *d* into the *dimension* field of *def*.
put_def_kind copies *k* into the *kind* field of *def*, if *k* equals *const_kind*, *proc_kind* or *tdecl_kind* the *readonly_access* bit of *def* is set, else it is cleared.
put_def_type copies *t* into the *type* field of *def*.

See also **types**
def_ptr, e_access, e_kind, e_type (Module idscope)

Check actions

Name check_access, check_definition, check_dim, check_kind, check_type, chk_def_access, chk_def_dim, chk_def_kind, chk_def_type, pointer_check, positive_check, result_access, result_definition, result_kind, result_type, zero_check,.

Definition **actions**

```
check_access      ( in a1: e_access; in a2: e_access;
                    in equal: boolean).
```

```
check_definition  ( in def1: def_ptr;
                    in def2: def_ptr;
                    in equal: boolean).
check_dim         ( in d1: integer; in d2: integer;
                    in equal: boolean).
check_kind        ( in k1: e_kind; in k2: e_kind;
                    in equal: boolean).
check_type        ( in t1: e_type; in t2: e_type;
                    in equal: boolean).
chk_def_access    ( in a: e_access; in def: def_ptr;
                    in equal: boolean).
chk_def_dim       ( in d: integer; in def: def_ptr;
                    in equal: boolean).
chk_def_kind      ( in k: e_kind; in def: def_ptr;
                    in equal: boolean).
chk_def_type      ( in t: e_type; in def: def_ptr;
                    in equal: boolean ).
pointer_check     ( in defp: def_ptr;
                    in def: def_ptr).
positive_check    (in nr: integer; in k: e_kind).
result_access     (out ra: e_access;
                    in a: e_access; in def: def_ptr).
result_definition (out rd: def_ptr; in def1: def_ptr;
                    in def2: def_ptr).
result_kind       (out rk: e_kind; in k: e_kind;
                    in def: def_ptr).
result_type       (out rt: e_type; in t: e_type;
                    in def: def_ptr).
zero_check        (in nr: integer; in k: e_kind).
```

Description The *check_** actions report an error message if their first argument does (not) match their second argument.

check_access reports the error message **Access conflict: *a1* ↔ *a2*** if *equal* is *true* and the *access* values *a1* and *a2* are not equal.
If *equal* is *false* and *a1* equals *a2* the error message **Access *a1* not allowed here** is reported.

check_definition() compares two definition records *def1* and *def2* and reports an error message if *equal* is *true* and *def1* and *def2* are not equal or if *equal* is *false* and *def1* and *def2* are equal.
Two definition records are considered to be equal if their types, (derived) kinds and dimensions are equal.
If *def1* or *def2* equals *undeclared* no comparisons are made and no errors reported.

check_dim reports the error message **Dimension conflict: *d1* ↔ *d2*** if *equal* is *true* and the integer values *d1* and *d2* are not equal.
If *equal* is *false* and *d1* equals *d2*, the error message **Dimension *d1* not allowed here** is reported.

check_kind reports the error message **Kind conflict: *k1* ↔ *k2*** if *equal* is *true* and the *kind* values *k1* and *k2* are not equal.
If *equal* is *false* and *k1* equals *k2*, the error message **Kind *k1* not allowed here** is reported.

check_type reports the error message **Type conflict: *t1* ↔ *t2*** if *equal* is *true* and the *type* values *t1* and *t2* are not equal.
If *equal* is *false* and *t1* equals *t2*, the error message **Type *t1* not allowed here** is reported.

The *chk_def_** actions report an error message if the first argument does (not) match the corresponding field of the definition record pointed to by *def*.

chk_def_access calls *check_access(a, a2, equal)* with *a2* being the *access* field of the definition record pointed to by *def*.
chk_def_dim calls *check_dim(d, d2, equal)* with *d2* being the *dimension* field of the definition record pointed to by *def*.
chk_def_kind calls *check_kind(k, k2, equal)* with *k2* being the *kind* field of the definition record pointed to by *def*.
chk_def_type calls *check_type(t, t2, equal)* with *t2* being the *type* field of the definition record pointed to by *def*.

pointer_check reports the error message **Possible dangling pointer reference** if the kind of *defp* equals *pointer_kind* and the scope level (data segment number) of *defp* is less than the scope level of *def*.

positive_check is used to check the number of arguments/indices of a procedure call/array. If *nr* is greater than 0 an error message: **Too few <items>** is reported, if *k* equals *proc_kind* **<items>** stands for **arguments** else it stands for **indices**.

result_access returns in parameter *ra* the value *a*, if *a* matches the *access* field of *def*. Otherwise, *ra* is assigned the value *unknown_access* and the error message **Access conflict: *a1* ↔ *a2*** is reported.
result_definition calls the action *check_definition()* to compare the definition records *def1* and *def2*. Afterwards *delete_definition()* is called to delete *def2*. The name field of *def1* (if set) will be cleared

and *rd* is assigned the value of *def1*. This action can be used if *def1* and *def2* are used as attribute containers to pass on type, kind and dimension in one call, instead of assigning them separately. **Note**: If either *def1* or *def2* equals *undeclared* no checks are performed and the output parameter *rd* is assigned *undeclared*.

result_kind returns in parameter *rk* the value *k*, if *k* matches the *kind* field of *def*. Otherwise, *rk* is assigned the value *unknown_kind* and the error message **Kind conflict: *k1* ↔ *k2*** is reported.

result_type returns in parameter *rt* the type *t*, if *t* matches the *type* field of *def*. Otherwise, *rt* is assigned the value *unknown_type* and the error message **Type conflict: *t1* ↔ *t2*** is reported.

zero_check is used to check the number of arguments/indices of a procedure call/array. If *nr* is equal to 0 an error message: **Too many <items>** is reported, if *k* equals *proc_kind* **<items>** stands for **arguments** else it stands for **indices**.

See also

```
types
   def_ptr, e_access, e_kind, e_type    (Module idscope)
```

Call actions

Name arg_access, arg_check, first_parm, next_parm.

Definition

```
actions
   arg_access( out a: e_access;     in pardef: def_ptr).
   arg_check ( out argdef: def_ptr; in pardef: def_ptr).
   first_parm( out pardef: def_ptr; out npars: integer;
               in procdef: def_ptr).
   next_parm ( out pardef: def_ptr);
```

Description

arg_access returns in *a* the kind of access needed for an argument of a procedure call depending on the corresponding parameter definition *pardef*. It should be called before an argument is parsed. If the *kind* field of *pardef* equals the value *rpar_kind* then *a* is assigned the value *address_access*, else *a* is assigned the value *unknown_access*.

arg_check is used to check if an argument definition *argdef* is consistent with the corresponding parameter definition *pardef* of a procedure call. It should be called after an argument is parsed. The action *check_type()* is called to compare the type values of *argdef* and *pardef*.
If the access value of *argdef* is *readonly_access* and the kind value of *pardef* is *rpar_kind*, the error message: **Illegal reference argument, <name> is read-only** is reported.

first_parm returns in *pardef* the definition of the first parameter of the procedure definition *procdef*, if *procdef* has no parameters *pardef* is assigned the value *undeclared*. The number of parameters of *procdef* is returned in the output parameter *npars*.

next_parm returns in *pardef* (which is both an input and an output parameter of *next_parm*) the definition of the next parameter following *pardef*. If the last parameter definition has already been reached *pardef* is assigned the value *undeclared*.

Assignment actions

Name assign_access, assign_definition, assign_kind, assign_type, clear_access, clear_definition, clear_kind, clear_type.

Definition

```
actions
    assign_access     ( out dest: e_access; in src: e_access).
    assign_definition ( out dest: def_ptr; in src: def_ptr).
    assign_kind       ( out dest: e_kind; in src: e_kind).
    assign_type       ( out dest: e_type; in src: e_type).
    clear_access      ( out dest: e_access).
    clear_definition  ( out dest: def_ptr).
    clear_kind        ( out dest: e_kind).
    clear_type        ( out dest: e_type).
```

Description

assign_access assigns the *e_access* value *src* to *dest*.
assign_definition assigns the *def_ptr* value *src* to *dest*.
assign_kind assigns the *e_kind* value *src* to *dest*.
assign_type assigns the *e_type* value *src* to *dest*.
clear_access assigns *unknown_access* to *dest*.
clear_definition assigns *undeclared* to *dest*.
clear_kind assigns *unknown_kind* to *dest*.
clear_type assigns *unknown_type* to *dest*.

See also

```
types
    def_ptr, e_access, e_kind, e_type    (Module idscope)
```

Module *vimcode*

The API module *vimcode* defines actions for generating **VI**rtual **M**achine (VIM) instructions. See page 107 and further for a more detailed description of the VIM instruction set.
With the *emit** actions it is possible to generate VIM instructions. Each instruction is defined as an optional label, followed by an operation value, followed by zero or more arguments. The optional label is retrieved from the internal variable **CurLabel**, which can be set by the action emit_label().

The integer representation (int_repr*) actions are supplied to allow indirect usage of the types *boolean*, *char* and *string* (because VIM code only knows the type *integer*). There are also some actions concerning manipulation and generation of the (segment) length table and the string table.

Instruction set

Name operation.

Definition
```
types
   operation =(
      abs_, add, and, call, crseg, descr, dlseg, dvi,
      eq, eqn, ge, gt, halt, jiff, jift, jump, ldcon,
      ldind, ldnvar, ldvar, ldxvar, le, lt, mdl, mul,
      ne, neg, nen, noop, not, or, pop, popn, rdbool,
      rdchar, rdint, rdstring, return_, stind, stnvar,
      stvar, stxvar, sub, swap, varaddr, wrbool, wrchar,
      wrint, wrstring, xvaraddr
   ).
```

Description The type *operation* defines the operation values of the VIM instruction set recognised by the VIM code interpreter. See page 107 and further for more information about the instruction set.

Initialise and finalise actions

Name initialise_vimcode, finalise_vimcode.

Definition
```
actions
   initialise_vimcode   ().
   finalise_vimcode     ().
```

Description *initialise_vimcode* initialises the generation of VIM instructions. An output file is opened, which will contain the generated instructions. The output filename consists of the input filename with the extension **.vim**. If the output file is already open the error message **VIM code already initialised** is reported.
finalise_vimcode generates the *halt* instruction. The length table and the string table are written to the output file, and the output file is closed. If the output file was not open the error message **VIM code not initialised or already finalised** is reported.

See also
```
actions
   get_index(), enter_length(), int_repr_string()
                                          (Module vimcode)
```

Assignment actions

Name assign_op, clear_op.

Definition
```
actions
```

```
assign_op  (out dest:operation; in src:operation).
clear_op   (out dest:operation).
```

Description The action *assign_op* assigns the operation value *src* to *dest*. The action *clear_op* assigns the operation value *noop* (= no operation) to *dest*.

See also **types**

operation (Module vimcode)

Operation actions

Name dy_op_bool, dy_op_char, dy_op_int, dy_op_real, mon_op_bool, mon_op_char, mon_op_int, mon_op_real.

Definition **actions**

```
dy_op_bool  ( out dest: boolean;
              in op: operation;  in src: boolean).
dy_op_char  ( out dest: char;
              in op: operation;  in src: char).
dy_op_int   ( out dest: integer;
              in op: operation;  in src: integer).
dy_op_real  ( out dest: real;
              in op: operation;  in src: real).
mon_op_bool ( out dest: boolean; in op: operation ).
mon_op_char ( out dest: char;    in op: operation ).
mon_op_int  ( out dest: integer; in op: operation ).
mon_op_real ( out dest: real;    in op: operation ).
```

Description The operation actions can be used to perform some arithmetic operations at compile-time. If the specified operation *op* cannot be performed for the specified type of operands the *mon_op*()* actions leave *dest* unchanged and the *dy_op*()* actions copy the value of *src* into *dest*. If operation *op* is monadic the *dy_op*()* actions perform *op* on *src* and the result is stored in *dest*.

dy_op_bool performs the dyadic operation *op* on *dest* and *src* and stores the boolean typed result in *dest*.
dy_op_char performs the dyadic operation *op* on *dest* and *src* and stores the character typed result in *dest*.
dy_op_int performs the dyadic operation *op* on *dest* and *src* and stores the integer typed result in *dest*.
dy_op_real performs the dyadic operation *op* on *dest* and *src* and stores the real typed result in *dest*.
mon_op_bool performs the monadic operation *op* on *dest* and stores the boolean typed result in *dest*.
mon_op_char performs the monadic operation *op* on *dest* and stores the character typed result in *dest*.

mon_op_int performs the monadic operation *op* on *dest* and stores the integer typed result in *dest*.
mon_op_real performs the monadic operation *op* on *dest* and stores the real typed result in *dest*.

See also **types**
operation (Module vimcode)

Checked instruction generation actions

Name emit, emit_call, emit_jump, emit_opn, update_dpl.

Definition
```
actions
    emit       ( in op: operation).
    emit_call  ( in def: def_ptr).
    emit_jump  ( in op: operation; in labelnr:integer).
    emit_opn   ( in op: operation;    in def:def_ptr).
    update_dpl ( out newdpl: integer; in def: def_ptr).
```

Description *emit* is used to generate parameter-less instructions. If operation *op* requires an argument, the error message **Cannot emit operation *'op'* without arguments** is reported.

emit_call is used to generate the *call* instruction. The argument of the *call* instruction is the value of the *dva* field of the definition record pointed to by *def*. *emit_call* itself calls *chk_def_kind(proc_kind, def, true)* to check whether *def* points to a procedure definition.
If *def* equals one of the constants *undeclared* or *null_def*, this action has no effect.

emit_jump is used for generating a jump operation (*jump, jiff* and *jift*). The argument of the jump operation, the integer parameter *labelnr*, should represent a unique label obtained by the action *get_label()*. If *op* is not a jump operation, the error message **Illegal argument of emit_jump(): *'op'*** is reported.

emit_opn is used to generate either the 0-dimensional or the n-dimensional version of the operator *op* depending on the dimension field of the definition record pointed to by *def*.
If *def* defines a 0-dimensional identifier the operator *op* is generated by calling *emit(op)*.
If *def* defines a multi-dimensional identifier, *ldcon* instruction(s) are generated to load the bound values of *def* (or to load 0 if *def* defines a dynamic array) on the stack, followed by the n-dimensional version of *op*.

The instructions *eqn, ldnvar, nen, popn* and *stnvar* are defined as the n-dimensional version of the *eq, ldvar, ne, pop* and *stvar* instructions.
If *op* has no n-dimensional equivalent instruction the error message **No multi-dimensional version of operation '*op*' available** is reported.

update_dpl assigns the sum of the *dva* field of the definition record pointed to by *def* and the size of the identifier defined by *def* to the output parameter *newdpl*.
The size of the identifier defined by *def* is derived recursively. If *def* defines a 0-dimensional identifier the size is 1. If *def* defines an *n*-dimensional identifier with a bound pair list the size is derived from the bound pair list. If *def* defines an *n*-dimensional identifier without a bound pair list (i.e. a dynamic array) the size is 2**n*+2.

See also

```
types
   operation                          (Module vimcode)
   def_ptr                            (Module idscope)
actions
   chk_def_kind()                     (Module idscope)
   get_label()                        (Module vimcode)
```

Load instruction generation actions

Name emit_ldcon, emit_load, int_repr_bool, int_repr_char, int_repr_string.

Definition

```
actions
   emit_ldcon      ( in arg: integer).
   emit_load       ( in def: def_ptr;   in a: e_access;
                     in setcheck: boolean).
   int_repr_bool   ( out dest: integer;
                     in src: boolean).
   int_repr_char   ( out dest: integer; in src: char).
   int_repr_string ( out dest: integer; in src: string).
```

Description *emit_ldcon* generates the *load constant* (*ldcon*) instruction with the value of *arg* as an argument. Only integer values can be passed as an argument to the *ldcon* instruction. boolean, character or string values are to be converted to an integer representation by calling the corresponding *int_repr_** action.

emit_load generates the instructions *ldcon, ldind, ldnvar, ldvar, ldxvar* or *varaddr* depending on access value *a* and the contents of the definition record pointed to by *def*. If *a* is unequal to *unknown_access* it overrules the value of the access field of *def*.

If the resulting access value equals *address_access* the variable address instruction *varaddr* or *xvaraddr* is generated. However if *def* defines a read-only identifier/expression, the error message **Expression/identifier is read-only, cannot be accessed by address** is reported.
If the resulting access value equals *indirect_access* a *load indirect* (*ldind*) instruction is generated. If *def* defines an indexed array identifier an *xvaraddr* instruction should be generated first (see also *emit_xvaraddr()*).
If *def* defines no pointer nor a reference parameter nor a array variable, the error message **Expression/identifier cannot be accessed indirectly** is reported.
If the boolean parameter *setcheck* equals **true** and *def* is not "being set" (that is: it has not been an argument of *emit_store()*), the warning message **Variable <name> is used before set** is reported.
If *def* defines a multi-dimensional identifier *emit_opn(ldvar, def)* is called, the *ldnvar* instruction will then be generated preceded by *ldcon* instruction(s) to load the bound values of *def* (or to load 0 if *def* defines a dynamic array) on the stack.
Records should be defined as 1-dimensional identifiers having one bound pair with lower bound 0 and upper bound equal to the number of fields (see also *add_def_bound()*).
See also the occurrence tables in the ***idscope*** section for the correct field values of a definition record.

int_repr_bool converts the boolean value *src* into its corresponding (ordinal) integer representation (*false*= 0, *true*= 1) *dest*.

int_repr_char converts the (ASCII) character value *src* into its corresponding (ordinal) integer representation *dest*.

int_repr_string inserts the string value *src* into the string table and assigns the corresponding index to the integer value *dest*. If there are no free indices left in the string table the error message **String table full** is reported.
The string table will be added to the generated VIM code the moment *finalise_vimcode()* is called.

See also

types

def_ptr	(Module idscope)
e_access	(Module idscope)

actions

add_def_bound()	(Module idscope)
emit_opn()	(Module vimcode)
emit_xvaraddr()	(Module vimcode)

```
emit_load()                          (Module vimcode)
finalise_vimcode()                   (Module vimcode)
```

Store instruction generation actions

Name emit_stargs, emit_store. emit_store

Definition

```
actions
    emit_stargs ( in def: def_ptr).
    emit_store  ( in def: def_ptr;   in reload: boolean).
```

Description

emit_stargs should be called to store the arguments of a procedure call (which are on top of the stack, in reverse order) into the data segment of the procedure.

emit_stargs calls *emit_store* for each parameter definition of the procedure definition record pointed to by *def* (in reverse order). Before *emit_store* is called the access field of a reference parameter definition is set to *direct_access*. Afterwards the access field is (re)set to *indirect_access*.
If the *kind* field of *def* is not equal to *proc_kind* or if *def* has no child definitions this action has no effect.

emit_store generates the instructions *stind, stnvar, stvar* or *stxvar* depending on the contents of the definition record pointed to by *def*.
If *def* defines a read-only identifier/expression, the error message **Cannot store, expression/identifier is read-only** is reported.
If the access field of *def* is *address_access*, the error message **Cannot change address of expression/identifier** is reported.
If *def* defines a multi-dimensional identifier *emit_opn(stvar, def)* is called, the *stnvar* instruction will then be generated preceded by *ldcon* instruction(s) to load the bound values of *def* (or to load 0 if *def* defines a dynamic array) on the stack.
Records should be defined as 1-dimensional identifiers having one bound pair with lower bound 0 and upper bound equal to the number of fields (see also *add_def_bound()*).
If *reload* is *true*, *emit_load()* is called after generation of the store instructions; this can be useful when an assignment is parsed which should leave a resulting value on the stack.
See also the occurrence tables in the ***idscope*** section for the correct field values of a definition record.

See also

```
types
    def_ptr                          (Module idscope)
actions
    add_def_bound()                  (Module idscope)
```

```
emit_opn()                                  (Module vimcode)
emit_load()                                 (Module vimcode)
```

I/O instruction generation actions

Name emit_read, emit_write.

Definition

```
actions
    emit_read  (in t: e_type).
    emit_write (in t: e_type).
```

Description *emit_read* generates one of the VIM read instructions *rdbool*, *rdchar*, *rdint* or *rdstring* depending on the type value of *t*. If type *t* is not supported by a VIM instruction the error message **No VIM read instruction for type '*t*'** is reported. Because no arguments are required, it is also possible to use *emit()* to generate a read instruction.

emit_write generates one of the VIM write instructions *wrbool*, *wrchar*, *wrint* or *wrstring* depending on the type value of *t*. If type *t* is not supported by a VIM instruction the error message **No VIM write instruction for type '*t*'** is reported. Because no arguments are required, it is also possible to use *emit()* to generate a write instruction.

See also

```
types
    e_type                                  (Module idscope)
actions
    emit()                                  (Module vimcode)
```

Segment instruction generation actions

Name emit_crseg, enter_length, get_index.

Definition

```
actions
    emit_crseg     (in level, index: integer).
    enter_length   (in index, value: integer).
    get_index      (out index: integer).
```

Description The moment a *create segment* (*crseg*) instruction should be generated, the length of a segment is not known. By reserving one entry for each segment in the so called length table the length of a segment can be stored at this entry the moment it is determined. As soon as a *crseg* instruction needs to be generated an entry in this table can be allocated by calling *get_index()*. The obtained index is then used as an argument to the *crseg* instruction, which can be generated by calling *emit_crseg()*. The moment the length of a segment is determined, it can be stored at the reserved entry by calling *enter_length()*. Finally, the complete length table will be

added to the generated VIM code when *finalise_vimcode()* is called.

emit_crseg generates the *create segment* (*crseg*) instruction with the specified scope level *level* and length table index *index* as an argument.

enter_length stores *value* in the length table at position *index*.

get_index returns the next free entry in the length table in parameter *index*. If the length table is full, the error message **Length table full** is reported.

See also **actions**
`finalise_vimcode()` (Module vimcode)

Label generation actions

Name emit_label, get_label.

Definition
```
actions
    emit_label (in labelnr: integer).
    get_label  (out labelnr: integer).
```

Description *emit_label* stores the value of *labelnr* in the internal variable **CurLabel**. The instruction being generated after *emit_label()* is called will be preceded by the current value of **CurLabel**.
If *emit_label()* is called more than once without any instruction being generated in between, *noop* instruction(s) will be generated preceded by the corresponding value(s) of **CurLabel**.

get_label returns the next (free) label number in *labelnr*. The first label number returned is 1, following label numbers are obtained by incrementing the label number obtained by the last call to *get_label()*.

See also **types**
`operation` (Module vimcode)

Array instruction generation actions

Name emit_bound, emit_descr, emit_swap_pop, emit_xvaraddr.

Definition
```
actions
    emit_bound   ( in i: integer; in def: def_ptr).
    emit_descr   ( in sn, dpl, dim: integer).
    emit_swap_pop( in def: def_ptr;
                   in truthval: boolean; in a: e_access).
    emit_xvaraddr( in def: def_ptr).
```

Description

emit_bound generates *ldcon* instructions to load a bound pair of *def* on the stack. Which bound pair is loaded depends on the value of *i*, if *i* equals the dimension field of *def* the first bound pair of *def* is loaded, if *i* equals 0 the last bound pair of *def* is loaded. If *def* has no bound pair or the value of *i* is out of range, this action has no effect.
emit_bound is to be called when parsing the indices of an indexed (static) array variable (dynamic array definitions do not have any bound pairs because the bounds of dynamic arrays are not known at compile-time).

emit_descr generates the *descriptor* (*descr)* instruction with the value of *sn*, *dpl* and *dim* as arguments. It should be called when a dynamic array declaration is parsed.

emit_swap_pop generates a *swap* and a *pop* instruction depending on the value of *a* and the contents of the definition record pointed to by *def*.
If *a* is unequal to *unknown_access* it overrules the value of the access field of *def*. If *truthval* equals **true** and the resulting access value is not equal to *address_access* and *def* has access value *indirect_access* the swap and pop instructions are generated.
emit_swap_pop is to be called when an address of an (indexed) variable is to be removed from the stack.

emit_xvaraddr generates a *ldcon* and a *xvaraddr* instruction. The *ldcon* instruction is generated to load the dimension field of *def* (or -1 if *def* defines a dynamic array) on the stack. This action is to be called when an indexed array variable is parsed.

See also

types

```
    def_ptr                              (Module idscope)
```

actions

```
    add_def_bound()                      (Module idscope)
```